TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

HANNU TUOMISTO

REAALIAIKAISEN TILANNEKUVAN JAKAMISEN SUORITUSKYKY

Diplomityö

Tarkastaja: Professori Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa
13. tammikuuta 2016

TIIVISTELMÄ

HANNU TUOMISTO: Reaaliaikaisen tilannekuvan jakamisen suorituskyky
Tampereen teknillinen yliopisto
Diplomityö, 44 sivua
Marraskuu 2016
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Pervasive Systems
Tarkastaja: Professori Hannu-Matti Järvinen

Avainsanat: Reaaliaikaisuus, suorituskyky, Meteor, Web tekniikat

Meteor on täyden pinon web-sovelluskehys, joka on omiaan yhteisöllisesti tuotetun informaation jakamiseen reaaliaikaisesti käyttäjiensä kesken. Työn tavoitteena oli tutkia jo valmiiksi aloitetun Meteor-pohjaisen tilannekuvasovelluksen suorituskykyongelmia ja ratkaista niitä käyttäen tekniikoita, joilla työ oli jo aloitettu tai joita on mahdollista liittää siihen ilman suurempia ongelmia. Toissijaisesti työssä oli tarkoitus tutkia vaihtoehtoisia web-tekniikoita, joiden avulla suorituskykyä voidaan parantaa.

Tilannekuvasovelluksessa osallistuvat henkilöt liikuttavat hallinnoimiaan objekteja ja välittävät objektiensa liikkeen reaaliaikaisesti toisilleen niin, että kukin osallistuja näkee reaaliaikaisena objektien sijainnin ja suunnan kertovan tilannekuvan, jossa objektit liikkuvat nykimättä ruudulla.

Keskeisenä ongelmana tilannekuvasovelluksessa olivat tilannekuvan välittämiseen ja esittämiseen liittyvät viiveet. Nämä asettavat haasteita sille, että käyttäjä kokee käyttöliittymän reagoivan nopeasti tehtyihin muutoksiin, ja sille, että osallistujien näkymät ovat yhdenmukaisia. Työssä perehdyttiin renderöinnin ja kommunikaation taustalla oleviin web-tekniikoihin ja niiden suorituskykyyn.

Riittävä reaaliaikaisuus tilannekuvasovelluksessa saavutettiin ohittamalla paikkaviestien välittäminen MongoDB-tietokannan kautta käyttämällä vain viestipalvelimen välimuistia ja koostamalla välitettäviä viestejä palvelinpäässä tietyllä aikavälillä, jolloin välitettävien viestien määrä olennaisesti väheni. Suorituskyvyn parantamiseksi mahdollisina kehityssuuntina evaluoitiin myös CSS3-, SVG-, canvas- ja WebGL-piirtotekniikoiden käyttöä eri mobiililaitealustoilla: testeissä WebGL oli yksiselitteisesti nopein, muilla menetelmillä tulokset olivat ristiriitaisia eri laitealustoilla.

ABSTRACT

HANNU TUOMISTO: Performance of sharing real-time snapshot
Tampere University of Technology
Master of Science Thesis, 44 pages
November 2016
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Professor Hannu-Matti Järvinen

Keywords: Real-time, performance, Meteor, Web techniques

Meteor is a full-stack web-application framework which is well suited for sharing community created information between users. The aim of the work is to investigate performance problems of a commenced Meteor-based application and solve them by applying techniques already used in the application or techniques which are easy to integrate in to it. A secondary goal of the work is to investigate alternative web-technologies to improve performance.

Participants of the snapshot application move their own controlled objects and transmit the movement of the objects to each other so that every participant can see the snapshot which displays the position and the direction of the objects and the movement of the objects on the screen without any jank.

Delays in transmitting and displaying the snapshot were central problems in the snapshot application. These delays cause challenges in perceiving the user interface reactive enough to user actions and in providing uniform view for the participants.

Sufficient perception of real-time could be achieved in the application by bypassing the usage of MongoDB-database in the location messages by using only memory of the message server and aggregating all the messages to be sent by the server during a certain time-period. By doing so the number of messages was greatly reduced. To improve the performance alternative drawing techniques CSS3, SVG, canvas, and WebGL were also evaluated with different mobile platforms as possible development directions. In the tests WebGL was found undeniably the fastest technique while results with the other techniques were incoherent between different platforms.

ALKUSANAT

Tämä työ on tehty Devatus Oy:lle vuonna 2016.

Haluan kiittää työni ohjaajaa DI Mika Filanderia mielenkiintoisesta aiheesta ja saadusta palautteesta. Haluan myös kiittää diplomityön tarkastajaa professori Hannu-Matti Järvistä kommenteista, joiden avulla työ saavutti lopullisen muotonsa. Lisäksi haluan kiittää isääni DI Timo Tuomistoa lukuisista korjaus- ja lisäysehdotuksista.

Vaasassa, 20.11.2016

Hannu Tuomisto

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

AJAX	Asynchronous JavaScript And XML
CGI	Common Gateway Interface
CSS	Cascading Style Sheets
DDP	Distributed Data Protocol
DOM	Document Object Model
GLSL	OpenGL Shading Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MDG	Meteor Development Group
SSE	Server Sent Events
SVG	Scalable Vector Graphics
TCP/IP	Transmission Control Protocol / Internet Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
W3C	The World Wide Web Consortium
WebGL	Web Graphics Library
WebRTC	Web Real-Time Communication
WHATWG	Web Hypertext Application Technology Working Group
XHTML	eXtensible Hypertext Markup Language

1. JOHDANTO

Web-sovellukset ovat jatkuvasti siirtyneet lähemmäksi työpöytäsovelluksia; On päästy kauas ajoista, jolloin verkkosivun pienetkin muutokset haettiin palvelimelta lataamalla sivu uudelleen yhtenä kokonaisuutena käyttäjän selaimeen. Nykyaikaiset web-sovellukset kommunikoivat yhä useammin palvelimen kanssa. Web-sovelluksilta vaaditaan yhä lyhyempiä vasteaikoja ja työpöytäsovelluksista tuttuja ominaisuuksia.

Työssä tutkitaan, kuinka hyvin web-tekniikat soveltuvat toteuttamaan tilannekuvasovelluksen, jossa on välitettävä objektien sijainti ja suunta reaaliaikaisesti usean samaan istuntoon osallistuvan käyttäjän kesken. Vastaava tilanne löytyy myös monista verkkopeleistä sillä poikkeuksella, että sovellus ei ole peli, jossa kilpaillaan, vaan jossa tehdään yhteistyötä. Tavallisesti verkkopelit ovat toteutettu natiivisovelluksina, sillä web-sovellusten suorituskykyä ei ole pidetty yhtä hyvänä kuin natiivisovellusten [1]. Sovellus oli valittu toteutettavaksi selainpohjaisena, jotta se toimisi suoraan usealla käyttöjärjestelmällä. Työ toteutettiin osana vaasalaisen Devatus Oy:n ohjelmistoprojektia.

Sovelluksen kehittäminen oli aloitettu käyttäen Scalable Vector Graphics (SVG) -pohjaista grafiikkaa Meteor-sovelluskehyksessä. Työssä keskitytäänkin sovelluksen suorituskykyyn liittyvien tekniikoiden kuvaamiseen, suorituskykyongelmien kartoittamiseen ja sovellukseen tehtyihin korjauksiin, jolla ohjelman suorituskyky saatiin riittäväksi. Loppuosassa työssä evaluoidaan web-tekniikoita, joilla sovelluksen suorituskykyä voitaisiin edelleen parantaa, mutta joita ei muutostyön mittavuuden takia ole implementoitu lopulliseen sovellukseen. Luvussa 2 käydään läpi yleisesti eri web-tekniikoita ja niiden suorituskykyyn vaikuttavia seikkoja. Luvussa 3 keskitytään sovelluksessa käytettyihin kirjastoihin: Meteor ja D3.js. Luvussa 4 käydään läpi vaadittavaa suorituskykyä, luvussa 5 kuvataan järjestelmän toimintaperiaate tarkemmin ja kuinka se vertautuu verkkopeleissä käytettyihin tekniikoihin. Luvussa kuvataan myös suorituskyvyn arviointiin käytettävät järjestelyt ja kokeilut sekä läpi havaittujen ongelmien syitä sekä käydään läpi niihin löytyviä korjausvaihtoehtoja. Luvussa 6 arvioidaan sovelluksen toimintaa korjausten jälkeen ja pohditaan jatkokehitys-ajatuksia, joita ei tämän työn puitteissa ei ehditty tai pystytty toteuttamaan.

2. WEB-SOVELLUSTEKNIIKAT JA NIIDEN SUORITUSKYKY

Luvussa esitellään web-sovelluksien keskeisten tekniikoiden kehittyminen nykymuotoonsa ja keskeiset vaikuttavat standardit. Aliluvussa 2.1 kerrotaan standardointiin keskeisesti vaikuttavista toimijoista, web-tekniikoiden historiasta ja keskeisistä käsitteistä. Aliluvussa 2.2 pohditaan web-sivun ja sovelluksen eroa, esitellään keskeiset markkinoilla olevat selaintoteutukset ja niiden suhde laadittuihin standardeihin. Aliluvussa 2.3 käydään läpi keskeiset periaatteet, joilla web-sovellukset muistuttavat yhä enemmän toimintatavaltaan perinteisiä työpöytäsovelluksia. Aliluvussa 2.4 käsitellään tekniikoita, joilla käyttöliittymän ulkoasu pystytään sopeuttamaan fyysisesti eri kokoisille ja resoluutioltaan erilaisille näytöille. Aliluvussa 2.5 käsitellään palvelinpään tekniikoiden kehitystä. Aliluku myös käsittelee säiepohjaisten ja tapahtumapohjaisten arkkitehtuurien eroja pohjustaen samalla aliluvun 2.8 selainsovellusten yksisäikeisen tapahtumapohjaisen arkkitehtuurin mukanaan tuomien ongelmien ymmärtämistä. Aliluvussa 2.6 kerrotaan keskeisistä selaimen ja palvelimen välisistä kommunikointistandardeista. Aliluvussa 2.7 käsitellään keskeisiin standardeihin, jotka mahdollistavat tässäkin sovelluksessa vaadittavan grafiikan esittämisen nykyselaimissa. Aliluvussa 2.8 luodaan katsaus web-sovellusten ongelmiin verrattuna natiiviratkaisuihin.

2.1 Yleistä

Asiakaspään toteutuksen kolme keskeistä teknologiaa ovat Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) ja JavaScript [2].

HTML–merkintäkielellä rakennettu dokumentti koostuu elementeistä, joiden alkamista ja loppumista merkitään erilaisilla tageillä. Nämä tagit määrittelevät HTML–dokumentin sisällön, jonka selain tulkitsee ja näyttää oikein muotoiltuna.

HTML–merkintäkieli sai alkunsa, kun Tim Berners-Lee ehdotti vuonna 1990 tapaa jäsentää Cernin monimuotoista tietoa kokeista, laitteistosta ja henkilöstöstä [3]. Tästä projektista tulikin kuviteltua suositumpi, ja Berners-Leen perusti World Wide Web Consortiumin (W3C), joka nykyään ylläpitää suurta osaa Webin standardeista. W3C:n standardeilla tai määrittelyillä on erilaisia kehitysvaiheita, joita ovat esimerkiksi työversio (Working Draft), suosituskandidaatti (Candidate Recommendation), suositusehdotus (Proposed Recommendation) ja suositus (W3C Recommendation). Viimeisimmässä vaiheessa standardi on käynyt läpi kattavan arvioinnin ja testauksen niin teoreettisissa kuin käytännön tilanteissa. Suositus ilmaisee standardin valmiutta siirtyä käyttöön ja toimii vaatimuksena selainohjelmien toteuttajille [4].

W3C:n ylläpitämästä HTML-standardista julkaistiin monia suosituksia vuoteen 1999 asti. Tällöin W3C julkaisi version 4.01, joka jäi pitkäksi aikaa HTML-merkintäkielen viimeisimmäksi versioksi. W3C:n sisällä syntyi vuonna 2004 ristiriitoja suunnasta, johon kieltä oltiin viemässä. Kiistan seurauksena syntyi ryhmittymä nimeltä WHATWG (Web Hypertext Application Technology Working Group). W3C itse lähti kehittämään XHTML–standardia ja WHATWG alkoi työstää kahta standardia: Web Forms 2.0 ja Web Apps 1.0, jotka myöhemmin yhdistyivät yhdeksi spesifikaatioksi nimeltä HTML5. W3C:n luovuttua XHTML-kehittämisestä alkoivat nämä ryhmittymät kuitenkin tekemään jälleen yhteistyötä, jossa W3C pitää yllä staattisempaa ja versioitua standardia ja WHATWG kehittää elävää standardia, jota kutsutaan vain nimellä HTML. W3C:n ylläpitämä standardi HTML5 sai suosituksen 28.10.2014. Merkkikielen kuvauksen lisäksi HTML5 sisältää kielen elementteihin liittyvien ohjelmointirajapintojen kuvauksia.

Toinen W3C:n ylläpitämä standardi on CSS (Cascading Style Sheets). CSS on tyylisäännöstö, joka määrittää kuinka HTML-dokumentin elementit kuuluu esittää. Sen avulla voidaan määrittää helposti kaikki samankaltaisia elementtejä koskevat säännöt, kuten määrittelyn alaisten elementtien sisältämän tekstin koko. CSS:n avulla voidaan myös määrittää elementtien sijainti toisiinsa nähden tarkemmin kuin pelkkää HTML-kuvauskieltä käyttämällä. CSS:n avulla on mahdollista myös tunnistaa sivun avanneen laitteen ominaisuuksia kuten laitteen käyttämän näytön suuruus. Käytettävän laitteen näytön leveyden selvittäminen on tärkeä ominaisuus laitekokoon mukautuvien eli responsiivisten verkkosovellusten tekemisessä, mistä kerrotaan lisää aliluvussa 2.4. CSS:n avulla on mahdollista myös määritellä elementtien ominaisuuksia riippuen siitä, missä yhteydessä dokumentti on esitetty. Esimerkiksi teksti voidaan esittää suurempana näytöllä kuin paperilla tulostettuna.

CSS:n rajoitteiden poistamiseksi sen toiminnallisuutta on laajennettu esiprosessoitavilla kielillä kuten Stylus, LESS ja SASS. Niiden avulla voidaan kirjoittaa tyylittelysääntöjä käyttäen esimerkiksi muuttujia ominaisuuksien arvoina. Tämä piirre puuttuu kokonaan tavallisesta CSS:stä.

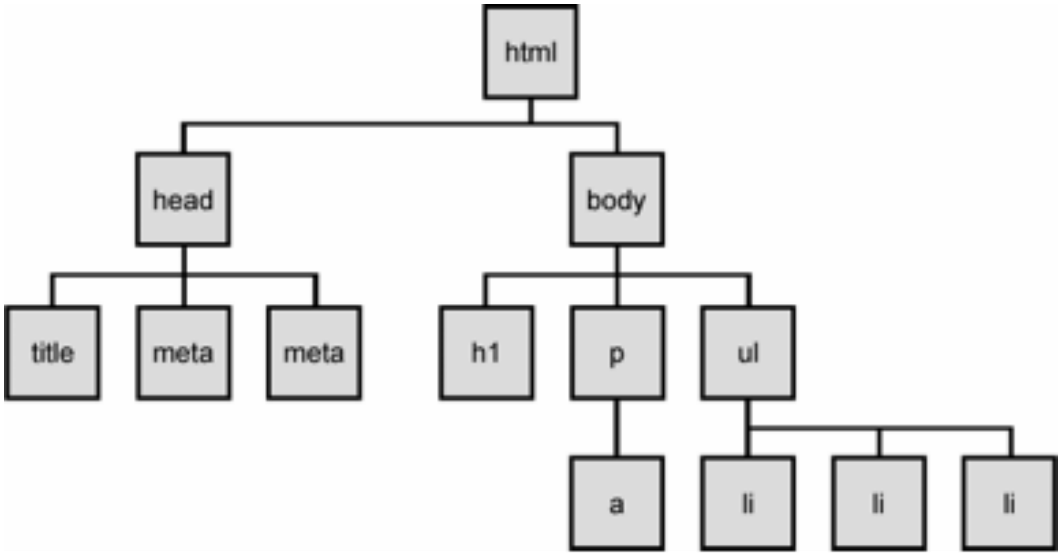
CSS oli versioon 2.1 asti yhtenäinen standardi, jota kaikki nykyselaimet tukevat hyvin. CSS3 on kuitenkin jaettu erillisiksi moduuleiksi, joiden kypsyysaste ja selaintuki vaihtelevat [5].

JavaScript on selaimien tukema ohjelmointikieli, jonka avulla voidaan toteuttaa interaktiivisuutta HTML-elementteihin. Web-teknologioiden kehittyessä JavaScriptin rooli on kasvanut jatkuvasti. Kaksi merkittävää standardia - ECMAScript ja DOM - ovat merkittävästi parantaneet JavaScriptin yhdenmukaisuutta eri selainten välillä.

ECMAScript määrittelee JavaScript-kielen perussyntaksin ja ydintoiminnot. Sen kehitys sai aikoinaan alkunsa Netscapen JavaScriptistä. Netscape kuitenkin luovutti vuonna 1996

JavaScriptin kehittämisen Ecma International -organisaatiolle, josta standardi sai nykyisen nimensä. Sekä JavaScript että Microsoftin aikoinaan tavaramerkkisyistä lanseeraama vastaava JScript pohjautuvat nykyisin tähän standardiin.

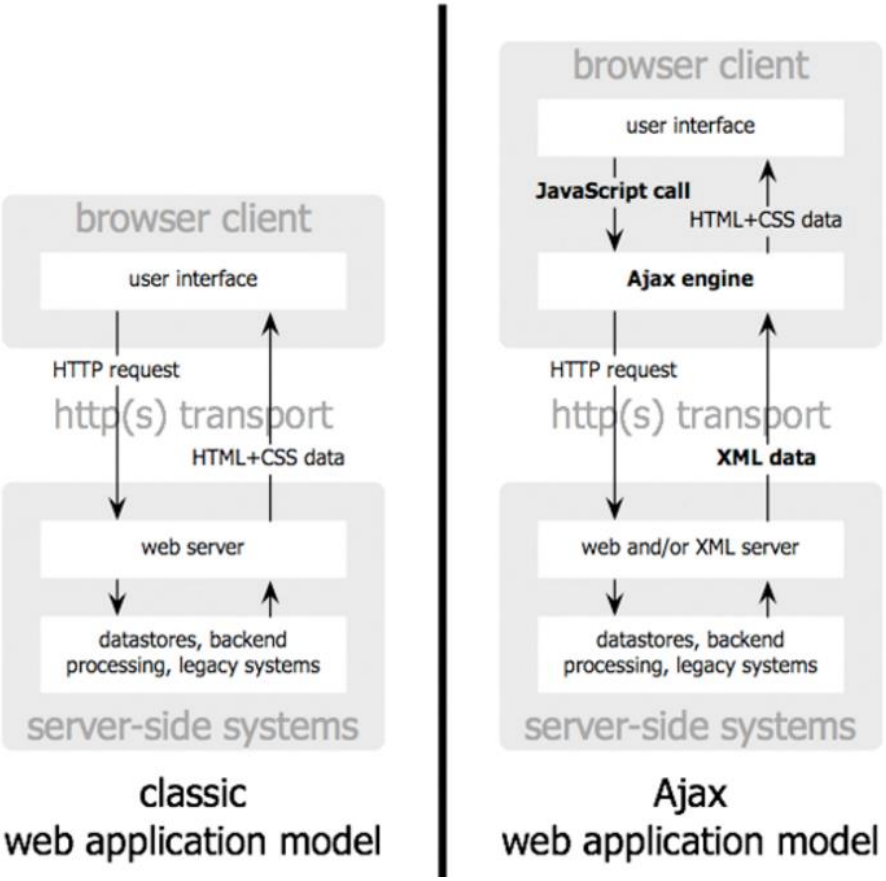
Samaan aikaan ECMAScriptin valmistumisen kanssa myös W3C sai valmiiksi dokumenttioliomallin (Document Object Model, DOM), jossa HTML elementit ja niiden attribuutit on määritelty olioina puurakenteeseen, jota havainnollistetaan kuvassa 1. DOM mahdollistaa HTML-dokumentin muokkaamisen DOMin tarjoaman rajapinnan kautta. Nämä muutokset sitten näkyvät HTML-dokumentissa. Jokaisen uuden DOM standardipäivityksen liitteenä on sen päivitetty JavaScript rajapintasidonta [6].



Kuva 1. *DOM-oliomalli [7].*

Selainten JavaScript onkin nähtävä koostuvaksi kolmesta osasta: ECMAScriptin muodostamasta ytimestä, DOM rajapinnasta sekä muista selaimen toimintaan liittyvien olioiden rajapinnoista. Näitä muita olioita - joko erikseen tai yhdessä DOMin kanssa, joskus vain pelkästään selainkäyttöliittymän näkyviin osiin liittyviin olioihin viitaten - kutsutaan usein yhteisnimellä Browser Object Model (BOM). Käsitteen sisältävillä rajapinnoilla ei ole taustalla yhtä selkeää koordinoivaa standardointiorganisaatioita [8].

Tärkeitä teknologioita JavaScriptin historiassa ovat olleet myös AJAX ja jQuery–kirjasto. AJAX tarkoittaa tekniikkaa tai pikemminkin joukkoa tekniikoita, joiden avulla palvelimelle voidaan tehdä kutsuja ja päivittää sivun sisältöä vain osittain lataamatta koko sivua uudestaan. Tätä toiminnallisuutta havainnollistetaan kuvassa 2. AJAX hyödyntää DOM-rajapintaa ja asynkronista XMLHttpRequest-funktiota. jQuery-kirjasto piilotti selainten eroavaisuuksia ja mahdollisti edistyneempien ominaisuuksien käytön yhdenmukaisesti [9].



Kuva 2. *Perinteinen sivun hakeminen (vasemmalla) ja AJAX (oikealla) [10].*

HTML:n, CSS:n ja JavaScriptin perinteinen vastuunjako on ajan saatossa hieman hämärtynyt. HTML5:n myötä HTML sai monia uusia ominaisuuksia kuten ääni- tai videotiedostojen toistaminen ja monet erityyppiset tiedonsyöttökentät, jotka ennen toteutettiin JavaScriptillä tai erillisillä selainliitännäisillä. Myös CSS on poikennut osittain alkuperäisestä tarkoituksestaan, ja sen avulla on mahdollista toteuttaa elementtien ominaisuuksien vaihtamista ajan suhteen eli animaatioita. Tätäkin pidettiin aikoinaan vain JavaScriptin tai erilaisten selaimen lisäosien tehtävänä. Viime aikoina suosittu JavaScript-sovelluskehys React on myös purkanut tätä vastuunjakoa käyttämällä esikääntäjässään JSX-syntaksia (kirjainlyhenne vailla tarkkaa englanninkielistä merkitystä), joka on ECMAScriptin syntaksilaajennus, jossa JavaScriptiä ja HTML-koodia kirjoitetaan sekaisin [11].

2.2 Web-sovellukset ja selaimet

Yksinkertaisimmillaan selain tekee HTTP-kutsun palvelimelle ja saa takaisin HTML-sivun, jonka sisällön selain tulkitsee [2]. Web-sivun rinnalle on ajan saatossa kehittynyt termi web-sovellus. Näiden termien erona pidetään sitä, että web-sovelluksen avulla on tarkoitus saada tehdyksi jokin tehtävä, kun taas perinteisiltä web-sivuilta vain haetaan tietoa. Eroa voidaan kuvata myös siten, että web-sovelluksissa tuotetaan tietoa, kun taas web-sivuilla tietoa vain kulutetaan [12].

Erilaisia selaimia on olemassa useita: suosituimmat ovat Google Chrome, Mozilla Firefox, Internet Explorer ja Safari. Selainten välinen suorituskyky renderöinnissä saattaa vaihdella suuresti, koska ne käyttävät erilaisia selainmoottoreita. Eri selaimilla myös monet muut ominaisuudet ovat toteutettu eri tavoilla tai ovat vielä toteuttamatta. Näistä eroavaisuuksista on saatavilla tietoa Caniuse-palvelussa [13]. Selaimen tukemia ominaisuuksia voidaan havaita myös Modernizr-kirjaston avulla ja mahdollisesti paikata havaittuja puutteita vastaavalla JavaScript-koodilla [14]. EcmaScript 2015 -standardin mukana tulleiden ominaisuuksien tukeminen eri selaimissa on myös vaihtelevaa, mutta näitä on mahdollista tunnistaa käyttämällä Babel-kirjastoa, joka osaa kääntää uutta syntaksia käyttävät ominaisuudet vanhempien selaimien tukemaksi syntaksiksi [15].

2.3 Yhden sivun web-sovellukset

Yhden sivun web-sovellukset (single page application, SPA) tarkoittavat tapaa toteuttaa web-sovellus, jossa verkkosivu ladataan kerralla kokonaan. Sen sijaan, että näytettäessä web-sovelluksissa uutta tietoa tarvitsisi aina ladata kokonaan uusi sivu, tehdään palvelimelle AJAX-kutsuja uudelle datalle, jonka perusteella nykyisen sivun DOMia voidaan muokata näyttämään uudet tiedot.

DOMia dynaamisesti muokkaamalla sovellukset saadaan vaikuttamaan käyttökokemukseltaan entistä sulavammilta, koska sivujen vaihdon välissä selain ei välähdä. Koska sivua ei jouduta hakemaan palvelimelta kokonaan uudestaan, tiedonsiirto on yleensä vähäisempää, jolloin palvelimen kuormituskin pienenee. Vaikkei käyttäjä vierailekaan enää eri sivuilla, HTML5:n myötä tullut History-rajapinta mahdollistaa myös yhden sivun web-sovelluksissa URL-osoitteen vaihdon ilman, että sivu joudutaan lataamaan uudestaan. Tämä onnistuu vaihtamalla ensin haluttu sisältö JavaScriptin avulla ja käyttämällä History-olion pushState-funktiota, jonka avulla voidaan vaihtaa selaimessa näkyvää osoitetta ja luoda historiatieto edellisestä sivusta. Tämän lisäksi tarvitaan kuuntelija popstate-tapahtumalle, jota kutsutaan, kun selaimen taakse- tai eteenpäin painikkeita painetaan [16].

2.4 Laitekokoon mukautuvat web-sovellukset

Mobiililaitteiden suosion kasvaessa tuli tarve miettiä verkkosivujen soveltuvuutta niiden näytöille. Aikaisemmin verkkosivuja oli suunniteltu vain pöytäkoneille ja tietyille näytön leveyksille, jotka olivat tavallisesti 800 tai 1024 pikseliä. Jos sivu piti esittää mobiililaitteilla, käytettiin ratkaisua, jossa mobiililaitteen tunnistaessaan palvelin lähettää pelkästään kyseenomaiselle laitteelle tehdyn sivun. Tällöin kuitenkin eri sivujen ylläpitäminen kävi työlääksi [17].

Laitekokoon mukautuvat eli responsiiviset sivut tunnistavat selaimen käyttämän näytön leveyden ja esittävät sisällön näytölle sopivassa muodossa. Esimerkiksi kapeammilla näytöillä voidaan normaalisti vierekkäin olevat sarakkeet näyttää allekkain. Mobiililaitteilla

käytetään nykyään hyvinkin suuria resoluutioita, jotka eivät ole suoraan verrannollisia näytön fyysiseen kokoon. Siksi laitteet ilmoittavat pikselisuhteen, joka on näytön resoluutio skaalattuna jollakin luvulla. Näin saadaan perinteisten monitorien kokoon vertautuva CSS-leveys [18].

2.5 Palvelinpuolen tekniikat

Web-tekniikoiden kehityksen alkuvaiheessa pelkillä selainpuolen tekniikoilla ei ollut mahdollista näyttää sivun osana dynaamisesti muuttuvaa tai laskettavaa tietoa, vaan siihen tarvittiin palvelimella suoritettavia skriptejä, joiden tulos näytettiin sivulla. Näistä tekniikoista ensimmäisiä oli CGI (Common Gateway Interface), jonka avulla tietyt URL-osoitteet (Uniform Resource Locator) voitiin määrittää palvelimella suoritettaviksi ohjelmina (tyypillisesti Perl- tai PHP- ohjelmointikielellä toteutettuja). Jokainen uusi CGI-palvelupyyntö kuitenkin käynnistää isäntäkoneessa uuden prosessin: suorittaminen tulee raskaaksi, varsinkin jos samanaikaisia pyyntöjä on useita.

FastCGI mahdollistaa (inter process communiction) socket-yhteyden luomisen saman palvelimen tai TCP-socket-yhteyden luomisen toisen palvelimen prosessiin, jolloin päästään eroon prosessi per pyyntö -mallista, ja sama prosessi voi palvella useita samanaikaisia pyyntöjä [19].

Alkuperäinen CGI-malli oli sopimaton myös Javalle, jossa uuden JVM:n käynnistäminen jokaista pyyntöä kohden ei ollut hitautensa takia ajateltavissa. Sun Microsystems kehittikin vastineeksi Java Servlet -spesifikaation: toiminnallisuus koostuu web-sovelluksia sisältävästä säiliöstä, joka isännöi ja suorittaa useita web-sovelluksia. Säiliö ylläpitää säievarastoa, jolle se lähettää tulevat palvelupyynnöt objekteineen (esimerkiksi HTTPServlet) suoritettavaksi. Monisäikeisyys parantaa olennaisesti suorituskykyä CGI:hin nähden.

Olennaisin kysymys palvelinpuolen tekniikoiden kehityksessä onkin ollut niin sanottu C10k-ongelma: miten palvella kymmeniä tuhansia samanaikaisia pyyntöjä [20]. Samanaikaisuuden haastetta voi periaatteessa ratkaista joko palvelimen sisäistä laitteistoa tai arkkitehtuuria kehittämällä tai kuorman jakamisella usean palvelimen kesken. Nämä ovat kuitenkin tämän työn fokuksen ulkopuolella. Seuraavassa käsitellään vain palvelimen sisäisiä ratkaisuja.

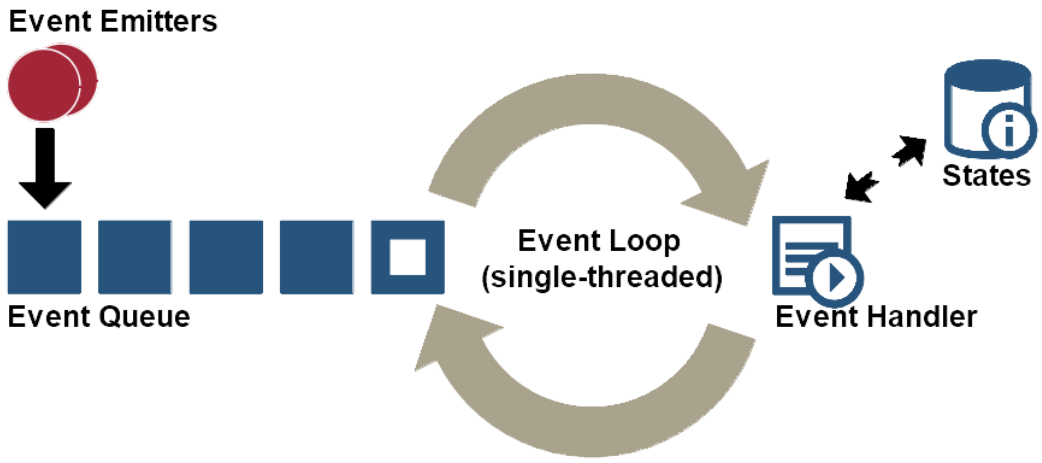
Palvelimen päätehtävä on suorittaa I/O pyyntöjä. Keskeisiä käsitteitä samanaikaisuuden suhteen ovat blokkaava tai blokkaamaton I/O ja asynkroninen tai synkroninen I/O, joita usein käytetään samanlaisina käsitepareina, vaikka niillä onkin hiuksenhieno ero: blokkaavissa tai blokkaamattomissa ratkaisuissa ohjelma kertoo alustalle, haluaako se vastauksen heti, vai palauttaako kutsuttava funktio ainoastaan kyselyn statuksen. Synkronisessa tai asynkronisessa kutsussa kutsuja joko säilyttää kutsun suorituksen kontrollin itsellään tai antaa sen pois kutsuttavalle funktiolle.

Toinen samanaikaisuuteen liittyvä termi on tehtävien ajettavien tehtävien priorisointi käyttäen joko irroittavaa vuoronnusta (pre-emptive scheduling) tai yhteistyövuoronnusta (co-operarative scheduling). Edellisessä järjestelmä ajaa mekaanisesti tietyillä aikasiivuilla korkeimman prioriteetin tehtäviä. Yhteistyövuoronnuksessa arkkitehtuuri luottaa siihen, että kukin tehtävä osaa itse vapauttaa oman suorituksensa.

Palvelinarkkitehtuureissa on ollut jo vuosikymmenet kaksi kilpailevaa linjaa: säiepohjaiset arkkitehtuurit ja tapahtumapohjaiset arkkitehtuurit: lisäulottuvuuden keskusteluun niiden paremmuudesta ovat tuoneet moniydinprosessorit, jotka mahdollistavat todellisen rinnakkaisprosessoinnin: yksiydinprosessoreilla rinnakkaisuus on vain näennäistä [19].

Säiepohjaisessa arkkitehtuurissa kukin kutsu käsitellään omassa säikeessään (säie-per-kutsu). Arkkitehtuuri on melko selkeä: esimerkiksi Apache MPM käyttää edelleen monisäieratkaisua. Usein säiepohjaisessa arkkitehtuurissa on lisäksi erillinen dispatcher-säie (acceptor-säie), jolla on käytettävissä säievarasto, josta se allokoi säikeen uudelle kyselylle. Jos vapaita säikeitä ei ole, uusia yhteyksiä ei muodosteta, ennen kuin säievarastosta vapautuu säie.

Tapahtumapohjaisessa arkkitehtuurissa yksi säie käsittelee useita yhteyksiä ja yhteyksiin liittyviä I/O-operaatioita (kuva 3).



Kuva 3. *Tapahtumapohjainen arkkitehtuuri: Tapahtumat niihin liittyvine funktioineen liitetään tapahtumajonoon [19]***.**

Tapahtumapohjaisissa arkkitehtuureissa tehtävien ajo pohjautuu tyypillisesti yhteistyövuoronnukseen, kun taas säiepohjaiset ratkaisut kayttävät irroittavaa vuoronnusta. Tapahtumapohjaisten arkkitehtuurien väitetään periaatteessa olevan säiepohjaisia nopeampia, joskin moniydinprosessorien hyödyntäminen on niillä vaikeampaa kuin säiepohjaisessa

arkkitehtuurissa. Suosittuja tapahtumapohjaisia palvelimia ovat muun muassa nginx (Github, Wordpress.com) ja lighthttpd (Youtube, Wikipedia) [19].

Node.js on myös hyvä esimerkki tapahtumapohjaisesta arkkitehtuurista: Node.js on joukko sidostoimintoja Chrome-selaimen käyttämän V8-JavaScript-moottorin ympärillä: socket-yhteydet, tiedostojen hallinta ja niin edelleen. Tämä rajapintojen joukko koostuu ainoastaan blokkaamatonta asynkronisista rajapinnoista. Node.js:llä on vain yksi pääsäie ja suorittava kutsupino. Node.js:n on sanottu yhdistävän sopivasti säiepohjaisten arkkitehtuurien pinon hallinnan helppouden ja tapahtumapohjaisten arkkitehtuurien tehtävien yhteistoiminnallisen suorituksen.

Useissa palvelinratkaisuissa tapahtumapohjainen ja säiepohjainen arkkitehtuuri on yhdistetty hybridiarkkitehtuuriksi, muun muassa Staged Event Driven Architecture (SEDA) on muodollinen kuvaus tällaisesta arkkitehtuurista.

Palvelinpuolen toteutukseen on historiallisesti käytetty eri ohjelmointikieliä kuten PHP, Python, Java, Visual Basic, C# ja Ruby. Näiden ohjelmointikielten avulla on toteutettu useita eri sovelluskehyksiä kuten Django (Python), ASP ja Ruby on Rails [21].

Node.js on haastanut vanhan asiakas-palvelin-mallin, jossa asiakas- ja palvelinpään osuudet on toteutettu eri ohjelmointikielillä. Node.js:n avulla on mahdollista kirjoittaa koko sovellus käyttäen JavaScriptiä.

Oma lukunsa ovat ohjelmointikielet, jotka on tarkoitettu rinnakkaisesti suoritettaviin tehtäviin: osa näistä, muun muassa Scala ja Clojure, on ajettavissa Java-virtuaalikoneessa. Erlang on aikoinaan Ericssonin kehittämä funktionaalinen ohjelmointikieli, joka siihen liittyvine Open Telecom Platform (OTP) -väliohjelmistoineen, -työkaluineen ja -kirjastoineen mahdollistaa rinnakkaisuuteen pohjautuvien palvelujen rakentamisen. Muun muassa WhatsApp käyttää Erlang-kieltä viestintäpalvelussaan mahdollistaen jopa kaksi miljoonaa yhtäaikaista käyttäjää yhdellä palvelimella.

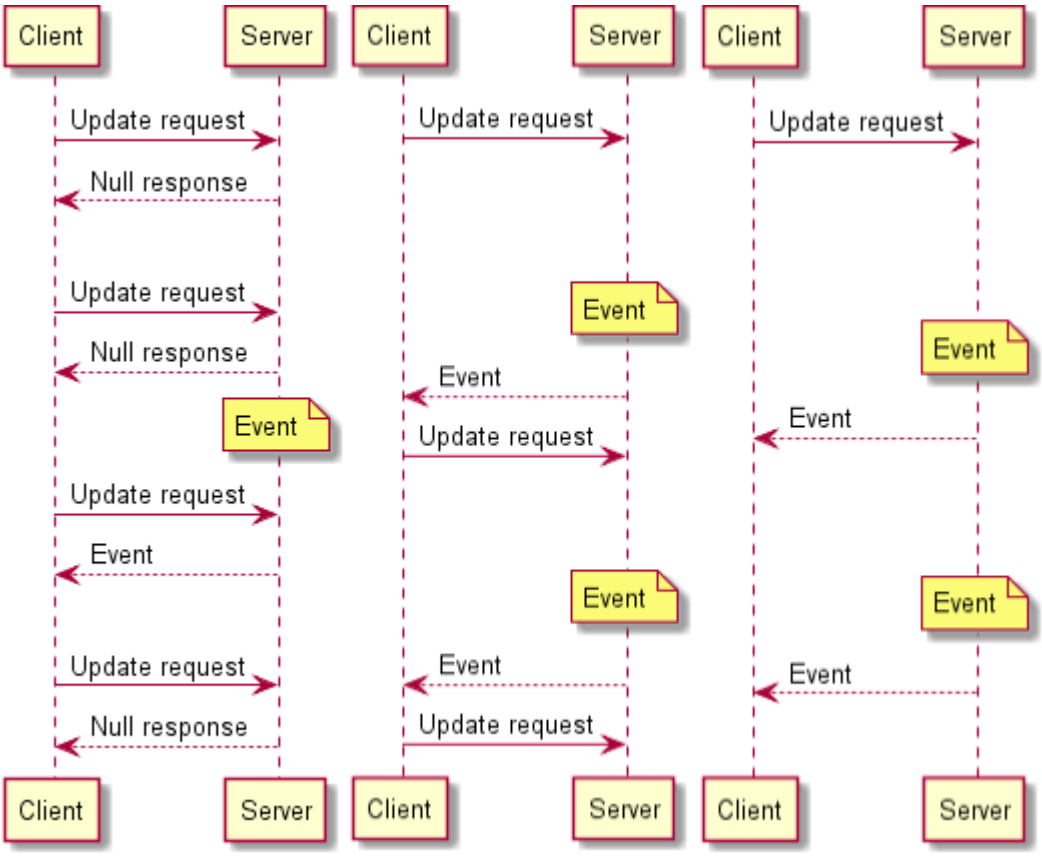
Kerrosarkkitehtuurimalli ja JavaScript mahdollistavat täysin uuden ajattelutavan työnjaossa selaimen ja palvelimen välillä. Uusimmat HTML5-tekniikat kuten Web Storage API ja IndexedDB mahdollistavat lähes koko sovelluksen siirtämisen asiakaspäähän. Esimerkiksi SPA-sovelluksessa palvelimelle jää puhtaimmillaan CSS-, HTML-, ja JavaScript-tiedostojen jakaminen muistuttaen web-palvelimien alkuperäistä tehtävää staattisten sivujen toimittajana. Nykyään palvelimien kuorma onkin usein siirtynyt reaaliaikainen tiedon jakamiseen esimerkeiksi peleissä ja chat-sovelluksissa. Näitä varten onkin usein kehitetty räätälöidyt palvelimensa.

2.6 Verkkoliikenne

Perinteisesti vuorovaikutus verkkosivujen kanssa on tarkoittanut käyttäjän tekemiä HTTP-kyselyjä palvelimelle ja jäämistä odottamaan, että palvelin palauttaa käyttäjälle kokonaan uuden sivun. Tämän tekniikan heikkoudet oivallettiin kuitenkin pian. Usein olisi riittänyt, että vain päivitetty data olisi siirretty selaimelle koko sivun sijaan, jolloin sivu päivitettäisiin vain muuttuneen datan suhteen. Tähän ratkaisuksi kehitettiin AJAX-tekniikka, joka mahdollisti yksittäisten elementtien sisällön päivittämisen ilman, että koko dokumenttia tarvitsee ladata uudestaan.

Verkkosivujen muutokset eivät kuitenkaan aina ole pelkästään käyttäjälähtöisiä, vaan myös palvelimella tapahtuvien muutosten halutaan välittyvän selaimelle. Usein tätä ongelmaa on ratkottu tekemällä kiertokyselyjä (polling) palvelimelle tietyin välein. Tämä kuitenkin aiheuttaa paljon turhaa liikennettä selaimen ja palvelimen välillä ja ratkaisuna on käytetty long polling -tekniikkaa, jossa selaimen ja palvelimen välille muodostetaan yhteys, jossa yhteys jää odottamaan palvelinpään tapahtumia tietyksi aikaa [22].

Edellisillä tekniikoilla on kuitenkin rajoitteensa, ja HTML5 tarjoaakin kahta uutta standardia tapahtumien välittämiseksi. Server Sent Events- (SSE) ja WebSocket-rajapinnat mahdollistavat palvelimella tapahtuvien muutosten välittämisen reaaliaikaisesti selaimelle. SSE toimii vain palvelimelta selaimelle, kun taas WebSocket on kaksisuuntainen. WebSocket-tekniikkaa käyttäessä selaimen ja palvelimen välille muodostetaan jatkuva yhteys. Kuvassa 4 on WebSocket-tekniikalla toteutettua kyselyä verrattu edellä mainittuihin kyselytekniikkoihin.



Kuva 4. *Vasemmalta oikealle: Ajax Polling, Ajax Long Polling ja WebSocket [22].*

Kuvasta on nähtävissä, että käyttämällä WebSocket-tekniikkaa saadaan vähennettyä palvelimelle tehtyjen kutsujen määrää ja samalla myös verkkoliikennettä. WebSocket vaatii jatkuvan yhteyden ylläpitämistä, mikä voi lisätä verkkoliikennettä, mutta yhteyden ylläpitäminen on kuitenkin kevyempää kuin jatkuvien turhien kyselyjen muodostaminen. WebSocket-tekniikka tarjoaa myös paremman vasteajan kuin kiertokysely vastauksen lähtiessä välittömästi eikä vasta seuraavan kyselyn saapuessa. Onkin mitattu, että WebSocket-tekniikan avulla voidaan verkkoliikennettä vähentää 500:1 suhteella ja latenssia vähentää kolmasosaan verrattuna perinteiseen pollaukseen [23]. Vaikka selain tukisikin WebSocket-tekniikkaa, eivät kaikki palvelimet sitä kuitenkaan aina tee, joten WebSocket vaatii tuen varasuunnitelmalle. Usein tämä ongelma ratkaistaan käyttämällä WebSocket-tekniikkaa erityisten kirjastojen kanssa, joilla on tuki korvaavalle teknologialle. Kirjastoista suosituimmat ovat Socket.IO ja SockJS [24].

WebSocket-protokolla mahdollistaa datan siirtämisen tekstinä ja binäärimuotoisena. Tekstinä data siirretään utf-8 muodossa. Binäärimuodossa implementoituja vaihtoehtoja ovat binany large object (blob) -raakadatamuoto ja WebGL:n inspiroimasta Typed Array -spesifikaatiosta lainatut ArrayBuffer- ja ArrayBufferView-tyypit. Kirjastojen tukemissa protokollissa on kuitenkin eroja, sillä aikaisemmin mainittu Socket.IO tukee nykyisin tekstimuodon lisäksi muotoja Buffer, Blob ArrayBuffer ja File, kun taas SockJS tukee vain tekstimuotoista dataa [25].

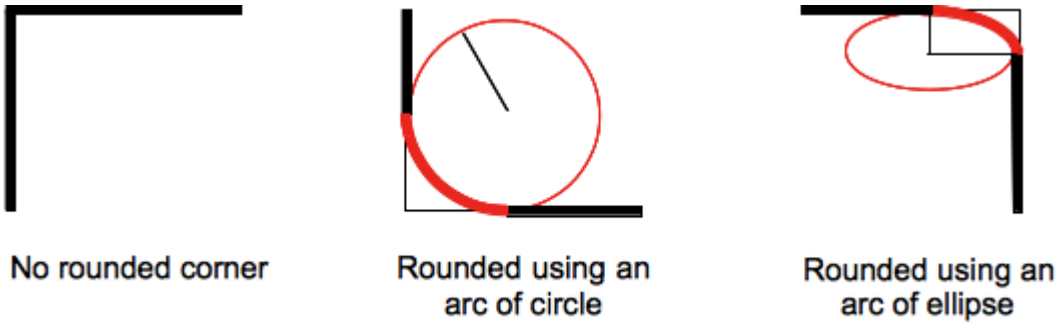
Viestinnän nopeuteen vaikuttaa oleellisesti myös siinä käytettävien viestien koko. Tyypillisesti WebSocket-viestinnässä käytetään JSON-objektin sarjallistamista. JSON-objekti koostuu nimi-arvo-pareista. Yksinkertaisin keinoin voidaan usein viestin kokoa merkittävästi pienentää. JSON-objektin sisältämät nimet kannattaa pitää mahdollisimman lyhyinä. Jos viestin JSON-objekti on vakiomuotoinen, voidaan JSON-objekti muuttaa JSON-taulukoksi, jolloin kunkin nimi-arvo-parin nimikenttä voidaan pudottaa kokonaan pois. Desimaalimuodossa oleva arvo voidaan sarjallistamalla muuttaa skaalaamalla kokonaisluvuksi ja vastaavasti viestiä purettaessa takaisin desimaalimuotoon. Muita potentiaalisia mahdollisuuksia Web Sockets -verkkoliikenteen suorituskyvyn parantamiseksi on välitettävän datakoon pienentäminen.

Edellä mainittujen tekniikoiden toiminta perustuu TCP/IP-protokollaan (Transmission Control Protocol / Internet Protocol), jossa varmistetaan, että lähetetty tieto saapuu vastaanottajalle oikeassa järjestyksessä ja vain yhteen kertaan. Oikea järjestys varmistuu, sillä jokainen lähetetty paketti sisältää järjestysnumeron, jonka mukaan vastaanottaja voi järjestää saamansa paketit. Viestin saapuminen perille varmistetaan sillä, että vastaanottaja kuittaa tiedonsiirrossa käytetyn paketin vastaanotetuksi. Jos kuittaus on puutteellinen tiettyjen kriteerien tai aikarajan puitteissa, paketti lähetään uudelleen, jolloin seuraavien pakettien lähettäminen viivästyy. Tämä saattaa muodostua ongelmaksi tilanteissa, joissa halutaan välittää tietoa reaaliaikaisesti [26].

Tämä ongelma on ratkaistavissa selaimissa käyttämällä WebRTC-rajapintaa (Web Real-Time Communication), joka pohjautuu UDP-protokollaan (User Datagram Protocol). UDP-protokolla ei vaadi vastaanottajan kuittausta paketin saapumisesta, jolloin pakettien lähettämisestä ei jäädä odottamaan, joten tiedonsiirto on nopeampaa. WebRTC- protokollan avulla voidaan päästä myös matalampiin latensseihin kuin WebSocket-tekniikkaa käyttämällä [27]. Protokollan heikkoutena on kuitenkin se, että on epävarmaa saapuvatko paketit lainkaan vastaanottajalle ja ovatko ne oikeassa järjestyksessä. WebRTC-rajapinnan käyttöönottoa haittaa myös se, että kaikki selaimet eivät tue sitä. Kirjoitushetkellä tuki puuttui Internet Explorer-, Edge- ja Safari-selaimilta [28].

2.7 Grafiikan piirtotekniikat

HTML5 ja CSS3 tarjoavat useita standardoituja ratkaisua 2D-grafiikan piirtämiseen. Osalla näistä tekniikoista voidaan tehdä myös animointia ilman JavaScriptiä, esimerkiksi CSS animaatioilla. HTML-perustekniikalla voidaan piirtää vain suorakaiteen muotoisia alueita. CSS3 mahdollistaa kuitenkin melko monipuoliset mahdollisuudet grafiikan esittämisen, vaikkakaan mielivaltaiseen grafiikkaan ei päästäkään: keskeiset CSS3-tyyliominaisuudet ovat elementin reunan säteen määräämä border-radius, läpinäkymättömyyttä kuvaava opacity ja elementin siirtymää osoittava transform. Border-radius mahdollistaa suorakulman kulmien pyöristämisen halutun säteiseksi ympyrän kaareksi tai jopa ellipsin osaksi hyödyntäen border-top, bottom-left tai right-radius -ominaisuuksia. Tätä havainnollistetaan kuvassa 5.



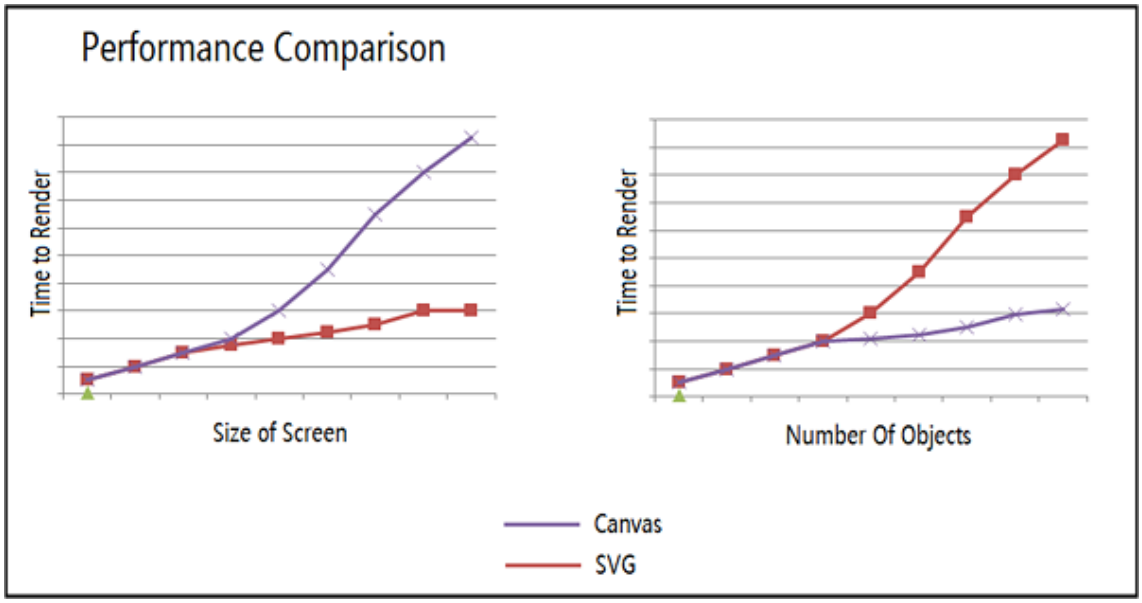
Kuva 5. *HTML-suorakaiteen pyöristäminen border-radius ominaisuudella [29].*

Transform-ominaisuus mahdollistaa HTML-elementin siirtymän (translation) haluttuun paikkaan ja kierron (rotation) haluttuun suuntaan. Transform-ominaisuudelle on olemassa sekä 2D- että 3D-versionsa. Tämän ominaisuuden 3D-tukeen liittyy useimmiten myös grafiikkaprosessorilla toteutettu laitteistokiihdytys. Opacity-ominaisuus mahdollistaa elementin läpinäkyvyyden säätämisen aseteltaessa elementtejä päällekkäin. Jos halutaan kuitenkin piirtää täysin vapaamuotoista grafiikkaa, on käytettävä SVG- ja canvas-rajapintoja.

SVG-rajapintaa on kehitetty jo vuodesta 1999 lähtien, mutta vasta HTML5:ssä se on osa standardia. SVG perustuu vektorigrafiikkaan, ja sen kuvaus on rakenteeltaan kuten HTML-elementit: sitä käytetään <svg>-tagien avulla HTML-koodin seassa, ja sitä voidaan muokata käyttäen JavaScriptiä ja CSS:ää. SVG:tä voidaan myös muokata sen oman DOM-rajapinnan kautta, joka kuitenkin eroaa HTML:n käyttämästä DOMista: SVG 1.1 pohjautuu DOM Level 2 CORE 2 spesifikaatioon toteuttaen sen määrittelemät rajapinnat, mutta ei esimerkiksi sisällä kaikkia DOM 2 -tapahtumia [30]. Myös selainten SVG-rajapinnan toteutuksissa on eroja, sillä esimerkiksi Internet Explorer- ja Edge-selaimilla SVG-elementteihin ei ole mahdollista kohdistaa CSS-animaatioita kuin vain ylimmän tason SVG-elementtiin [31].

Canvas on HTML-elementti, jolle voidaan ohjelmallisesti tuottaa graafista sisältöä tai animaatioita. Piirto tapahtuu pikselikohtaisesti JavaScript-pohjaisen rajapinnan kautta. Canvas soveltuu hyvin selainpohjaisten pelien toteuttamiseen, koska sillä on hyvin tunnettu matalan tason ohjelmointirajapinta, jossa useimmat selaimet osaavat hyödyntää käyttöjärjestelmänsä laitteistokiihdytystä. SVG:n vektoritekniikka soveltuu taas hyvin tilanteisiin, joissa kuvan on oltava tarkka eri näyttötarkkuuksilla. Siinä missä perinteinen kuvatiedosto pikselöityy kuvakokoa suurentaessa, skaalautuu vektorigrafiikalla luotu SVG-kuva aivan yhtä tarkaksi kuin alkuperäinenkin kuva. Pikselipohjaisessa grafiikassa taas skaalautumisongelma ratkaistaan usein tekemällä alkuperäisestä kuvasta tarpeeksi korkearesoluutioinen, jota sitten skaalataan alaspäin pienemmille resoluutioille. Nyrkkisääntönä voidaan pitää, että SVG renderöi isoja kuvia nopeammin kuin Canvas, mutta Canvas soveltuu paremmin suurien oliomäärien piirtämiseen. Tämä johtuu näiden rajapintojen luonteesta. SVG:n vektorigrafiikka skaalautuu hyvin mihin tahansa kokoon, mutta oliomäärän kasvu aiheuttaa suuren määrän DOM-elementtejä. Canvas taas vaatii

koko kuvan piirtämistä uudelleen, jos se halutaan päivittää [32]. Näitä Canvaksen ja SVG:n suorituskyvyn ominaisuuksia on havainnollistettu kuvassa 6.



Kuva 6. *Canvas ja SVG vertailua [32].*

Canvas-elementin kanssa voidaan myös käyttää WebGL-rajapintaa, joka mahdollistaa laitteistokiihdytetyn grafiikan piirtämisen. Se pohjautuu OpenGL ES 2.0 -rajapintaan hyödyntäen sen OpenGL Shading Language (GLSL) -varjostuskieltä [33], joka on tuetuissa selaimissa laitteistokiihdytetty. Selaimien tuki on ollut vaihtelevaa, mikä on rajoittanut sen yleistymistä, mutta kuitenkin kirjoitushetkellä WebGL on tuettu hyvin suosituimpien selaimien uusimmilla versioilla. WebGL:n kirjoittaminen sellaisenaan on kuitenkin melko työlästä ja vaatii 3D-grafiikan kanssa käytettävän matematiikan ymmärtämistä verrattuna SVG- tai pelkän canvas-pohjaisen grafiikan piirtämiseen. WebGL:n käyttämisen helpottamiseksi on kuitenkin kirjoitettu useita kirjastoja, jotka piilottavat WebGL:n monimutkaisuuksia yksinkertaisempien JavaScript-rajapintojen taakse. Näitä esimerkiksi 2D-grafiikkaan erikoistunut Pixi.js ja 3D-grafiikkaan Three.js [34].

2.8 Web-sovellusten rajoitteet

Sekä työpöytäselaimilla että mobiililaitteiden selaimilla on ongelmia pysyä uusien standardien tahdissa. Samoin standardien toteutukset selainten välillä saattavat poiketa toisistaan. Toteutustapojen erilaisuuden kiertämiseksi on käytetty muun muassa jQuery Mobile tai Kendo UI Mobile komponenttikirjastoja, jotka pohjautuvat HTML5-ratkaisuihin, ja joilla pyritään saamaan maksimaalinen yhteensopivuus selainten välillä. Ne pyrkivät myös tarjoamaan tavan toteuttaa käyttöliittymiä, jotka ovat reaktiivisia eli päivittyvät käyttäjävuorovaikutuksesta tai datasisällön muutoksista yhtä nopeasti kuin natiivisovellusten käyttöliittymät.

Mobiililaitteilla natiivisovellukset usein hyödyntävät laitteen antureita kuten kameraa, gyroskooppia tai yhteyksiä kuten Bluetooth ja NFC, mutta pelkillä web-sovelluksilla näihin on rajoitettu pääsyoikeus.

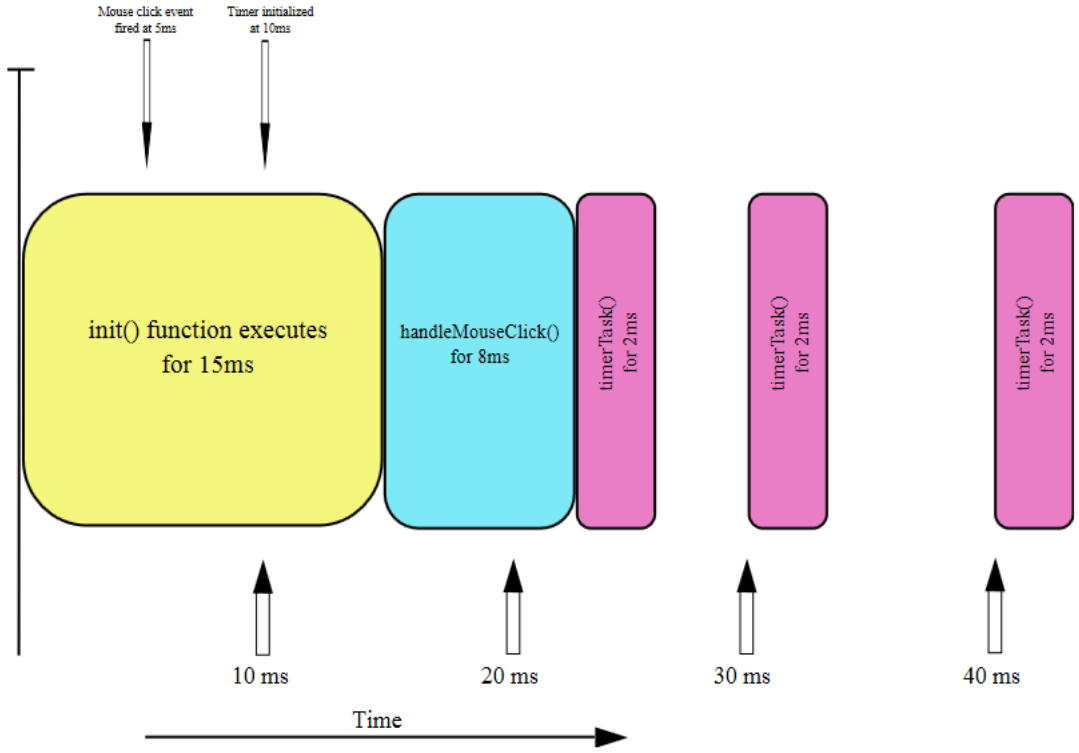
HTML5-pohjaisten ratkaisujen on todettu olevan käytettävyydeltään ja suorituskyvyltään heikompia verrattuna natiivisovelluksiin. Web-sovellukset vaihtavat usein sovelluksen sulavuuden helppoon siirrettävyyteen alustojen kesken. Suorituskykyerot natiivisovelluksen ja HTML5-pohjaisten sovellusten välillä kuitenkin jatkuvasti pienenee mobiililaitteiden laskentatehon kasvaessa [1].

JavaScript ei tue moniajoa natiivisti tai kirjastojenkaan avulla kuten perinteisemmät ohjelmointikielet Java ja C++. JavaScriptin samanaikaisuuden mallia (concurrency model) kutsutaan usein tapahtumasilmukaksi (event loop). Ohjelma suorittaa viestijonoon (message queue) lisättyjä tapahtumaviestejä siinä järjestyksessä, kuin selaimen eri toiminnot niitä generoivat.

Tapahtumia generoivat viestijonoon tyypillisesti DOM-tapahtumat (kuten ondrag ja onclick), asynkronisen XMLHttpRequest funktion paluuviestit ja asynkronisten ajastimien (setTimeout, setInterval) liittyvät tapahtumat.

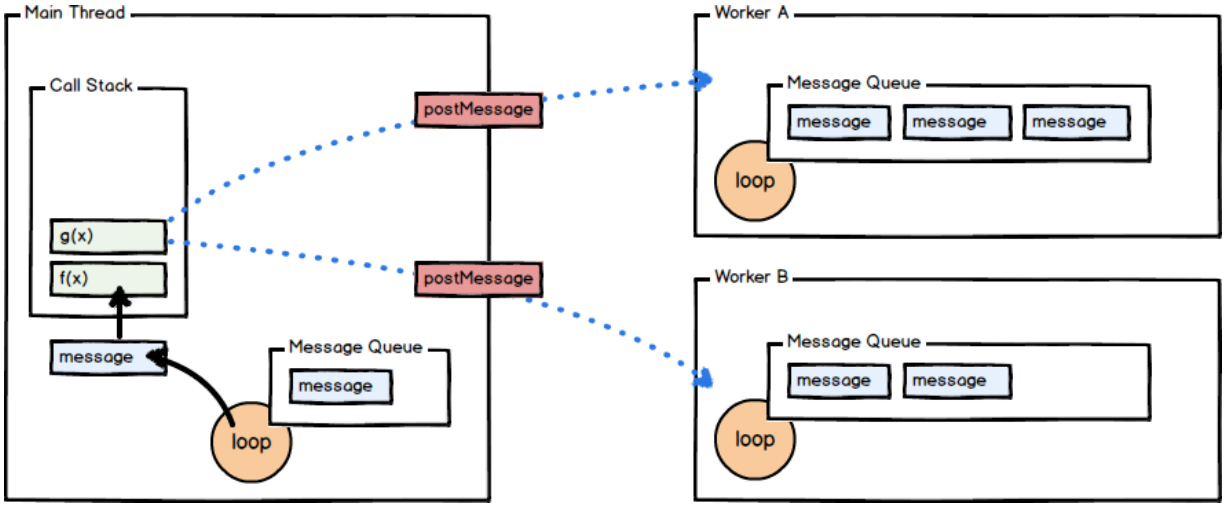
Tapahtumaviestit sisältävät suoritettavan takaisinkutsu-funktion (callback function), joka siirtyy argumentteineen erilliseen kehyspinoon (stack of frames). Jos kehyspinon funktio kutsuu toista funktiota, asetetaan tämä funktio päällimmäiseksi pinoon. Kun päällimmäinen funktio on saatu kokonaan suoritetuksi, se poistetaan pinosta. Vasta kun kehyspino on tyhjä, otetaan viestijonosta sinne seuraavaksi taltioitu viesti käsiteltäväksi. Eli vasta kun viesti on täydellisesti käsitelty, siirrytään käsittelemään seuraavaa viestiä. Asynkroniset funktiokutsut (esimerkiksi setInterval) siirtyvät pois pinosta, mutta niiden käsittely jatkuu selaimessa, kunnes niihin liittyvä tapahtuma laukeaa suoritettavaksi, jolloin tapahtuma ja viittaus siiheen liittyvään takaisinkutsufunktioon siirretään viestijonoon.

SetTimeout- tai setInterval-metodit takaavat vain sen, että tietty tapahtuma ja siihen liittyvä takaisinkutsufunktio tulee sijoitettua viestijonoon tietyn ajanjakson kuluttua. Jos viestijonossa on muita viestejä ennen kyseessä olevaa viestiä, niiden suoritus viivästyy, kunnes kaikki jonon edeltävät viestit on käsitelty (kuva 7). Kuvasta näkyy, että yksikin hidas funktiokutsu viivästää koko jonon kutsut johtaen muun muassa ajastimen takaisinkutsu-funktion viivästymiseen. Pahimmillaan raskas funktiokutsu voi johtaa koko sivuston näennäiseen pysähtymiseen.



Kuva 7. *Ajastimen viivästyminen toisen tapahtuman takia [35].*

Web Workers -rajapinnan avulla voidaan hyödyntää nykyään yleisiä moniydinprosessoreita ja suorittaa tehtäviä eri säikeissä. Tapaukset, joissa JavaScriptillä ladataan paljon dataa, ovat yleisiä. Web Workers -rajapinta mahdollistaa JavaScriptin ajamisen taustalla ilman, että muu sivusto joutuu odottelemaan sitä [35]. Kukin worker toimii omassa säikeessään noudattaen sisäisesti edellä kuvattua tapahtumasilmukkamallia. Tätä on havainnollistettu kuvassa 8. Workerit kommunikoivat pääsäikeen kanssa postMessage-funktion avulla, joka välittömästi asettaa vastaanottavan säikeen viestijonoon ko. funktion lähettämän tapahtuman. Vastaanottava säie (joko worker tai pääsäie) käyttää Worker.onmessage-tapahtumankäsittelijää määritelläkseen funktion, joka tapahtuma-viestiin assosioidaan; se tulee suoritetuksi, kun kyseessä oleva tapahtuma viestijonossa käsitellään.



Kuva 8. *Kommunikointi pääsäikeen ja workerin välillä tapahtuu postMessage-funktiokutsulla [36].*

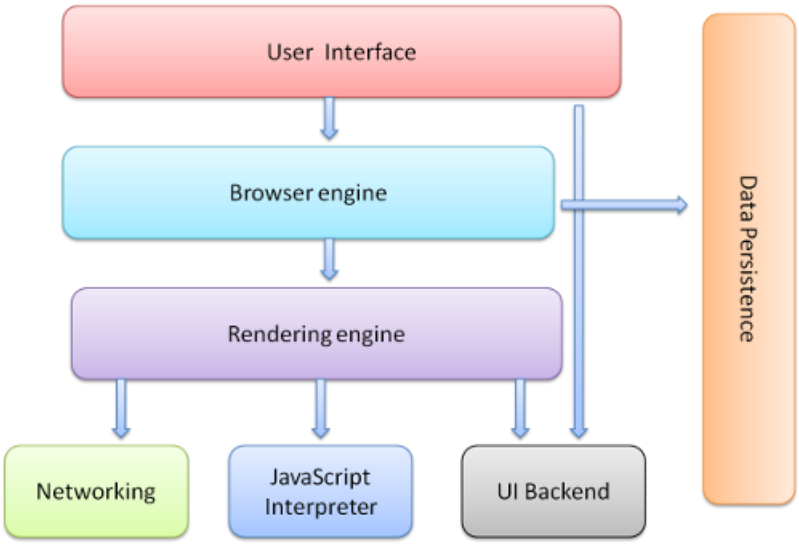
Web Workers -rajapinnan käyttö on kuitenkin raskasta niin prosessorin kuin muistinkin kannalta, eikä Web Workers -toteutuksia ole tarkoitettu käytettävän suuria määriä samanaikaisesti. Web Workers -rajapinta ei pääse myöskään vaikuttamaan selaimen DOM-rakenteeseen, jolloin sen toiminnallisuus on rajattua. Parhaiten ne soveltuvat tapauksiin, joissa suoritetaan raskaita laskutoimituksia ilman, että käyttöliittymä lukkiutuu tai tapauksiin, joissa syötteelle tehdään tarkistuksia ajonaikaisesti, esimerkiksi tekstikentän oikeinkirjoituksessa [37].

3. SIVUN RENDERÖINTI JA SUORITUSKYVYN MITTAAMINEN

Luvussa esitellään HTML-sivun piirtämiseen liittyvät yleiset vaiheet ja ongelmat sekä työkalut eri vaiheiden suorittamiseen liittyvän toiminnallisuuden ja virheiden seurantaan. Aliluvussa 3.1 kuvataan aluksi selainten yleinen rakenne, renderöintimoottorin rooli, yleisimmin käytetyt renderöintimoottorit ja yleiset renderöintimoottorien sivun rakentamiseen ja esittämiseen liittyvät aiheet. Aliluvussa 3.2 kuvataan haasteet ja ratkaisumallit, joilla sivun renderöintiin liittyvä kuvataajuus saadaan vaiheistettua näytön virkistystaajuuden kanssa. Aliluvussa 3.3 esitellään työkaluja, joilla on mahdollista jäljittää piirtämisen vaiheita ja toiminnallisuutta.

3.1 Selainten rakenne

Jotta web-sovelluksia voisi optimoida, on selainten toimintaperiaate ymmärrettävä. Selainten päätarkoitus on kuvantaminen eli renderöinti: selaimen renderöintimoottori tuottaa näkyvän esityksen esitettävän sivun HTML-elementeistä ja niihin liittyvistä CSS-ominaisuuksista. Nämä generoidaan suoraan HTML-tiedostoista ja CSS-tyylitiedostoista tai välillisesti JavaScriptin avulla. Käytännössä selain joutuu huolehtimaan myös verkkoliikenteestä, JavaScript-tulkinnasta, vuorovaikutuksesta käyttäjän kanssa sekä selaamiseen liittyvän tiedoista kuten selainhistoriasta, evästeistä, sivujen väliaikaisesta tallennuksesta. Selaimen yleinen rakenne on esitetty kuvassa 9.



Kuva 9. *Selaimen rakenne [38].*

Eri selaimet käyttävät eri renderöintimoottoreita: Internet Explorer käyttää Trident-, Firefox Gecko- ja Safari WebKit-pohjaista moottoria. Chrome ja Opera käyttävät Blink-

moottoria, joka on variaatio WebKit-moottorista. Windows 10 käyttöjärjestelmän Edge-oletusselaimen renderöintimoottori on EdgeHTML, joka on muunnos Trident-moottorista [39].

Yksittäisen sivun piirtäminen selaimessa voidaan JavaScriptin suorittamisen ja CSS-tyylien uudelleenlaskemisen jälkeen jakaa kolmeen eri vaiheeseen: asemointi (layout), maalaaminen ja koostaminen (compositing).

Asemoinnilla tarkoitetaan vaihetta, jossa selain laskee kunkin elementin vaatiman tilan ja sijainnin. Koska yhden elementin koko voi vaikuttaa toisen elementin kokoon mentäessä esimerkiksi DOM-puuta alaspäin, voi prosessi vaatia paljon laskentaa. Asemoinnista käytetään myös termiä uudelleenlaskeminen (reflow).

Asemointi tapahtuu automaattisesti CSS-tyylien laskemisen päätyttyä. Yleinen ongelma SPA-sovelluksissa on kuitenkin vuorottainen asemointiin liittyvien CSS-parametrien asettaminen ja uudelleen lukeminen, jolloin lukeminen käynnistää uudelleen asemoinnin. Hyvin moni DOM- tai SVG-ominaisuus voi aiheuttaa uudelleenasemoinnin [40]. Rinnakkaiselle tyylien laskennalle ja asemoinnille käytetään nimeä ”layout trashing”, ja se on yleisin syy dynaamisten web-sovellusten suorituksen hitauteen: ratkaisuna pidetään DOMin luku- ja kirjoitusoperaatioiden ryhmittämistä omiksi kokonaisuuksikseen [41]. Käytännössä koodin hajautuksen takia tämä ei ole helppoa [42].

Maalaamisvaiheessa piirretään tekstit, värit, kuvat ynnä muut visuaaliset osat elementeistä. Elementit jaetaan usein eri kerroksiin, jotka maalataan erikseen. Maalaamisvaihe voidaan lisäksi jakaa vielä itse piirtokäskyjonon rakentamiseen ja itse pikselien täyttämiseen. Jälkimmäistä vaihetta kutsutaan usein myös rasteroinniksi, ja se tapahtuu usein eri säikeessä kuin piirtäminen.

Koostamisvaiheessa eri kerroksiin maalatut kokonaisuudet yhdistetään, jolloin määritetään kuinka kerrosten elementit leikkaavat toisensa ja näkyvät toistensa päällä.

Jos dokumentissa vaihdetaan elementin geometriaan vaikuttavaa ominaisuutta kuten korkeutta, leveyttä tai sijaintia, täytyy selaimen laskea uudelleen kaikkien muiden elementtien sijainnit ja rakentaa sivu kokonaan uusiksi. Tällöin suoritetaan asemointi, maalaaminen ja koostaminen. Tämä on kaikkein raskain operaatio selaimen suorituskyvyn kannalta ja sitä pitäisi pyrkiä välttämään. Jos elementissä vaihdetaan vain maalattavaa ominaisuutta kuten taustakuvaa tai tekstin väriä, ei asemointia tarvitse tehdä. Kevyimpään tulokseen päästään, kun vältetään niiden ominaisuuksien muuttamista, jotka aiheuttavat uudelleenasemoinnin tai maalaamisen. Näitä ovat CSS:n *transform*-ominaisuudet, joita ovat muun muassa *translate*, jonka avulla voidaan siirtää elementtiä kerroksellaan. Ilman uudelleenasemoinnin tarvetta elementin koko voidaan lisäksi asettaa *scale*-omaisuudella ja elementin läpinäkyvyys *opacity*-ominaisuudella[43].

3.2 Sivun piirtämisen ajoitus

Jotta animaatiot näyttäisivät sulavilta, on animaation kuvataajuuden (tai kehysnopeuden) vastattava esittämiseen käytettävän monitorin tai näytön virkistystaajuutta. Usein tämä tarkoittaa 60 kuvaa sekunnissa, jolloin yksittäisen kuvan piirtämisen vaadittavien operaatioiden on tapahduttava 1/60 sekunnissa eli noin 16,7 millisekunnissa.

Jos kuvataajuus ja virkistystaajuus eivät vastaa toisiaan, on mahdollista, että päätelaitteen näkyvän kuvan perustana oleva näytönohjaimen näyttöpuskuri (frame buffer) koostuu kahdesta tai useammasta kuvasta. Tämä näkyy kuvassa vaakasuuntaisina siirtyminä, repeytyminä (tearing). Vaikka taajuudet olisivat jopa tismalleen samat, voi kuvataajuuden ja virkistystaajuuden vaihe-ero aiheuttaa sen, että kuva repeytyy vakiokohdassa. Repeytymistä korjaamaan on näytöissä pitkään käytetty pystytahdistusta (Vertical Sync, VSync), jolla monitori kertoo näytönohjaimelle, milloin se on valmis vastaanottamaan uuden näyttöpuskurin sisällön, eli näytönohjaimesta tulee tavallaan monitorin orja.

VSync-ominaisuuden aktivointi aiheuttaa maksimaalisen kuvataajuuden putoamisen näytön virkistystaajuudelle. Jos kuvataajuus on alhaisempi kuin virkistystaajuus, voi VSync aktivointi pahimmillaan johtaa näytönohjaimen kuvataajuuden ajoittainen puolittumiseen esimerkiksi 60:stä 30:een, joka näkyy ikään kuin nykimisenä (jank, stuttering) [44].

Ideaalista siis olisi, että uuden kehyksen renderöintiin kuluva aika olisi hieman alle virkistystaajuutta 60 hertsiä vastaavan ajan 16,7 millisekuntia. Käytännössä esimerkiksi pelisykliin kuluva aika on oltava maksimissaan 10 millisekuntia, koska selainta ja CPU:ta saattavat rasittaa myös muut rinnakkaiset tapahtumat (esimerkiksi roskienkeruu), jotka rasittavat suoritinta.

Aikaisemmin monissa tapauksissa animaatioita toteutettiin asettamalla 16 millisekunnin ajastin JavaScriptin setInterval- tai setTimeout-metodeilla. Koska ajastin ei tismalleen vastaa vaadittavaa aikaa, kuvan renderöinnin aloitus ajautuu epätahtiin kuvan virkistysajankohdan kanssa, jolloin kuva menetetään. Lisäksi ajastimen toiminnan vastatessa suurempaa kuvataajuutta kuin näytön virkistystaajuus, tehdään turhaa työtä kuvien piirtämiseksi, joita ei koskaan ehditä näyttää.

Vaikka ajoitus olisikin täysin tarkka, JavaScriptin ajastimet eivät ole täysin tarkkoja [45]. Selainten ja taustalla olevan käyttöjärjestelmän ajastimen tarkkuus on melko karkea: Windows XP:llä se oli aikoinaan oletusarvoisesti 1/64 s, eli periaatteessa JavaScriptin ajastusmetodit ovat tarkkoja vain lähimpään noin 15,625 millisekunnin välein toistuvaan ajankohtaan. Järjestelmän ajastustarkkuutta voidaan tosin muuttaa ohjelmallisesti, mitä useat ohjelmat käyttävätkin hyväkseen. Chrome käytti tätä aikoinaan hyväksi pudottaen ajastustarkkuuden yhteen millisekuntiin, mutta nostaen sen sitten 2 - 4 millisekuntiin: muutos on aina globaali, eli samaan aikaan pyörineen Firefox-selaimen ajastustarkkuus parani samalla. Sama rajoite koski myös JavaScriptin Date-metodia: sen palauttama aika

oli saman kelloajastimen viimeksi päivittämä aika, eli tarkka vain lähimpään kelloajastimen ajankohtaan.

Järjestelmän ajastimen taajuuden nostamisella on myös muita haittavaikutuksia: Se lisää keskeytyksiä hidastaen ohjelmien suoritusta ja kasvattaen laitteen tehonkulutusta [46].

JavaScriptin ajastimiin liittyviin tarkkuus- ja näytön virkistystaajuuden synkronointiongelmiin on kehitetty ratkaisuksi *requestAnimationFrame*-metodi, jonka avulla voidaan piirtää uusi kuva, kun edellinen on valmis tai kun näyttö pystyy sen näyttämään. Ohjelma 1 kuvaa rekursiivisesti kutsuttavan *requestAnimationFrame*-metodin käyttöä.

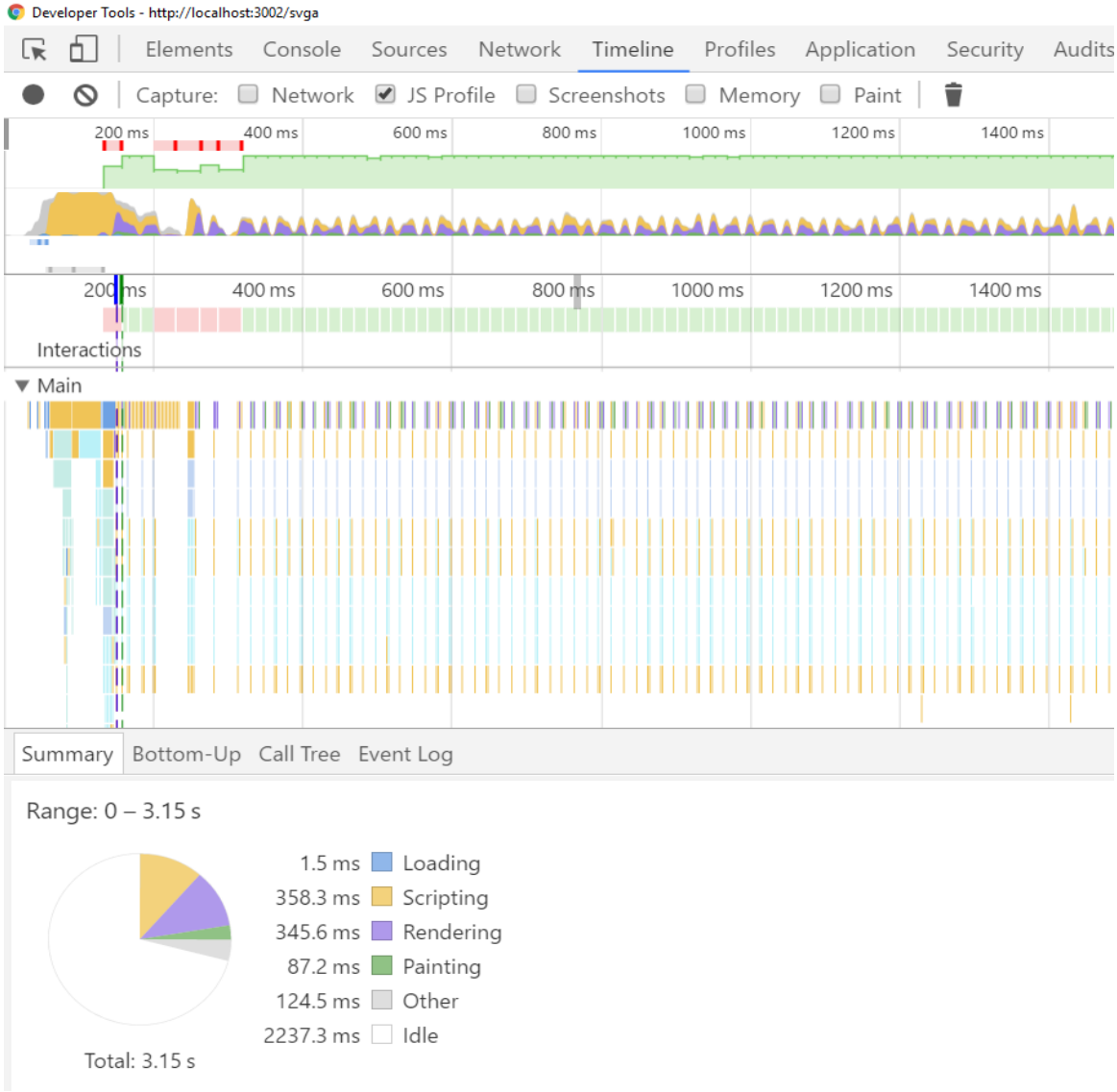
```
function update() {

    // yhden kuvan piirtämisen aikana tapahtuvan
    // muutoksen koodi tähän
    window.requestAnimationFrame(update);
}
window.requestAnimationFrame(update); // toistuvan päivityksen aloittaminen
```

Ohjelma 1. *Esimerkki rekursiivisesta requestAnimationFrame-metodin käytöstä.*

Tapauksissa, joissa laitteen laskentateho ei riitä renderöimään kuvaa virkistystaajuuden tahdissa, osaa *requestAnimationFrame* laskea kutsuväliä tasaisesti (esimerkiksi 30 kertaan sekunnissa). Lisäksi *requestAnimationFrame* on optimoitu pysähtymään, jos ikkuna, jossa sitä suoritetaan, on toisessa välilehdessä. Tällöin mobiililaitteita käytettäessä saadaan säästettyä akkua[47].

3.3 Selaimen suorituskyvyn mittaaminen

Verkkosivun suorituskyvystä on mahdollista saada tietoja työpöytäselaimilla käyttämällä selaimien kehittäjätyökaluja: niistä löytyy muun muassa timeline-ominaisuus, joka näyttää, mihin prosessoriaika kului tiettynä käyttäjän määrittämänä aikajaksona. Näistä tiedoista voidaan erotella JavaScriptin suorittamiseen kulunut aika JavaScript-funktiotasolla sekä saada tietoa, kuinka kauan aikaa kului asemointiin, maalaamiseen ja koostamiseen. Työkalut näyttävät myös kuvataajuuden ja korostavat kohdat, joissa ei päästy 60 kuvaa sekunnissa -kuvataajuuteen. Kuvassa 10 on nähtävissä tyypillinen Timeline-tilanne Chrome-selaimella (versiossa 52.0.2743.116).

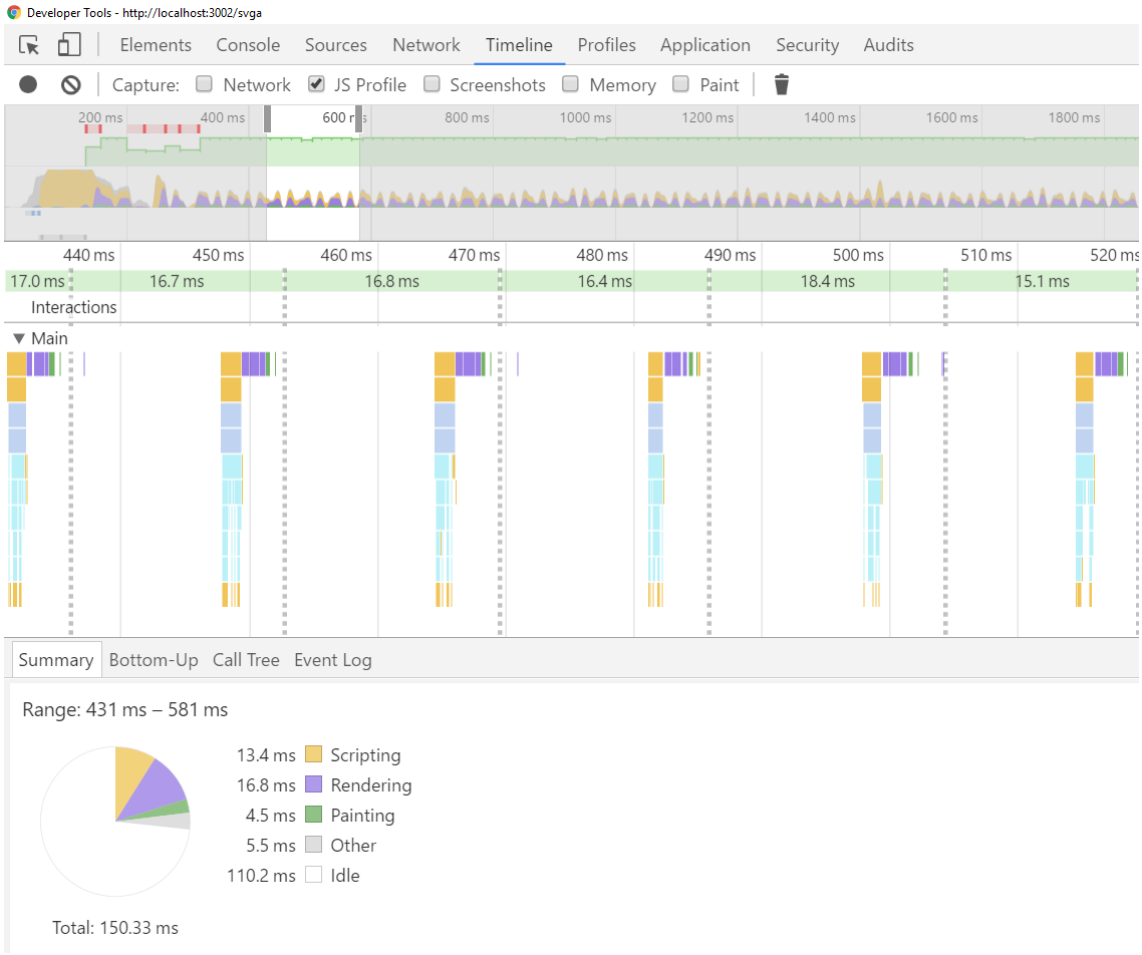


Kuva 10. *Timeline-työkalun perusnäkymä.*

Punaisella merkityt kohdat osoittavat ajankohdat, joissa kuvataajuus jäi selvästi alle kuudenkymmenen. Summary-osiossa voidaan nähdä, kuinka prosessoriaika jakautui mitatun ajanjakson aikana eri tyyppisten operaatioiden kesken ja kuinka paljon aikaa oltiin tekemättä mitään.

Chrome-selaimen timeline-työkalu käyttää yhteisnimeä rendering kuvaamaan CSS tyylien uudelleenlaskemista ja asemointia. Myös maalaus ja koostaminen on työkalussa niputettu painting-yhteisnimen alle.

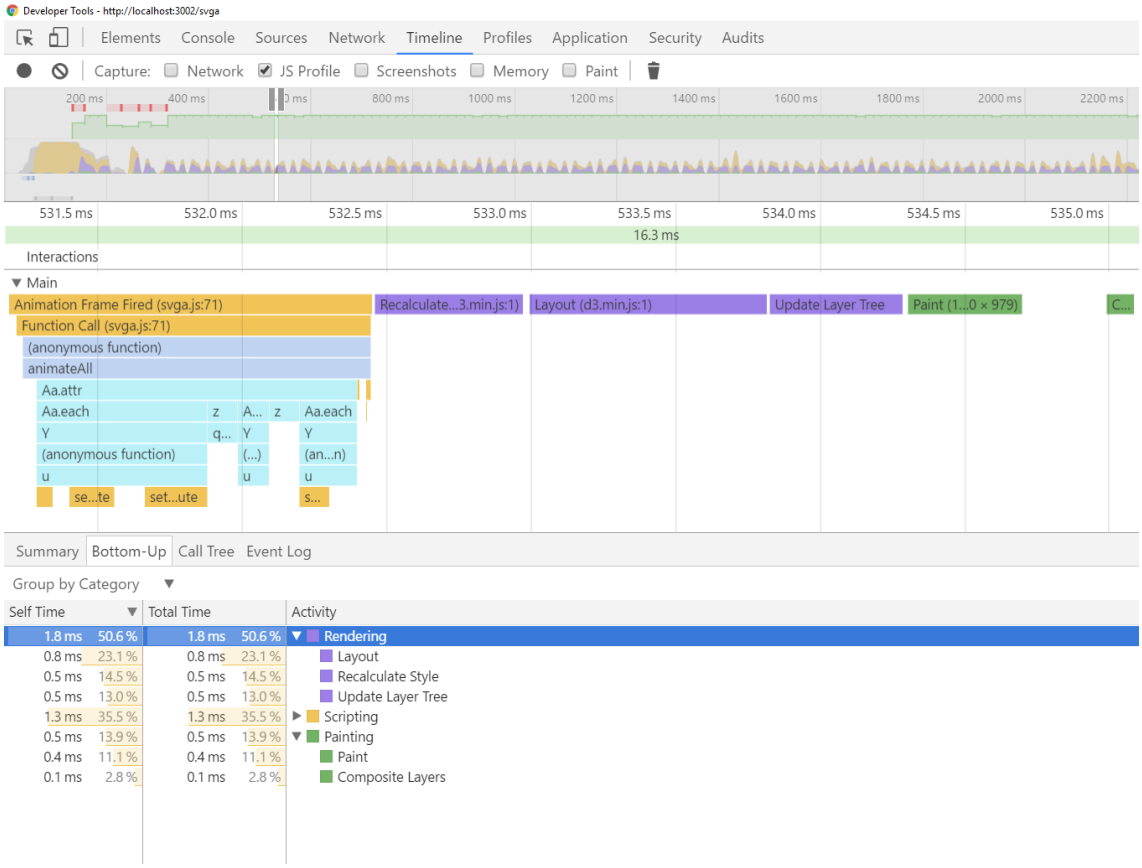
Koko aikajanan näyttävästä kuvasta on hankala nähdä erottaa yksittäisiä kuvia, joten kuvassa 11 on rajattu tarkasteltavaa aluetta.



Kuva 11. *Tarkasteltavan alueen rajaaminen Timeline-työkalulla.*

Kohdistaessa tarkkailua pienemmälle alueelle voidaan nähdä peräkkäiset kuvat ja niiden piirtämiseen kuluneet ajat. Kuvassa nähtävä 18.4 millisekunnin kuvan piirtäminen kestää kauemmin kuin tavoiteltava 16,7 millisekuntia, joten se näkyy myös ”lovena” ylemmässä osassa, joka näyttää valitun ajanjakson kuvataajuudet kyseisellä aikajaksolla.

Yksittäisen kuvan piirtämisen kulunut aika on nyt nähtävissä selkeämmin ja miten se jakautuu eri piirtämisen vaiheiden välille, mutta tarkkoja kuvauksia aikaa vievistä metodikutsuista ja operaatioista ei vielä voida erottaa. Kuvassa 12 onkin rajattu alue näyttämään vain yhtä kuvaa.



Kuva 12. *Yksittäisen kuvan operaatioiden tarkastelu Timeline-työkalulla.*

Kun näytettävä alue rajoitetaan yksittäiseen kuvaan, eri web-sivun renderöinnin vaiheet ja kutsuttavat metodit ovat selkeästi nähtävissä. Vaihtamalla Bottom-Up-välilehdelle on mahdollista nähdä, kuinka kuvan piirtämiseen kulunut aika jakaantuu eri renderöinti-operaatioiden ja JavaScriptin kesken.

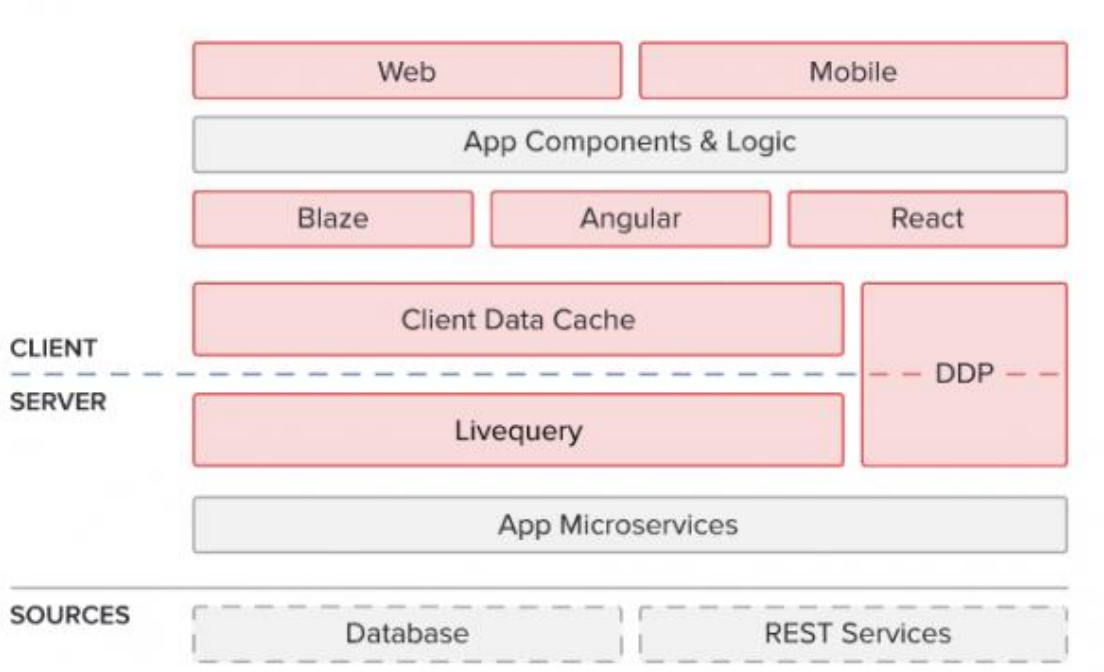
Kehittäjätyökaluja ei ole suoraan käytettävissä mobiililaitteiden selaimilla, mutta työkalujen avulla se on mahdollista pöytätietokoneeseen yhdistettynä. Android-laitteilla Chrome-selainta on mahdollista jäljittää (debug) etänä työpöytäkoneen Chrome-selaimen kautta ja navigoimalla osoitteeseen *chrome://inspect*. Tämän kautta voidaan käynnistää kehittäjätyökalut yhdistetylle laitteelle. Windows Phonen selainta on taas mahdollista profiloida etänä Visual Studion kanssa. IOS-laitteen Safari-selaimen profilointi taas on mahdollista yhdistettynä Mac-tietokoneeseen.

4. WEB-SOVELLUKSEN TEOSSA KÄYTETYT TEKNIIKAT

Luvussa esitellään keskeiset kirjastot ja kehitysympäristöt, johon tilannekuvasovellus perustuu. Aliluvussa 4.1 esitellään Meteor-sovelluskehys ja sen keskeisimmät toimintaperiaatteet. Aliluvussa 4.2 kerrotaan animoinnin tuottamiseen käytetystä D3.js-kirjastosta. Aliluvussa 4.3 kuvaillaan näiden tekniikoiden soveltamisen haasteet.

4.1 Meteor.js

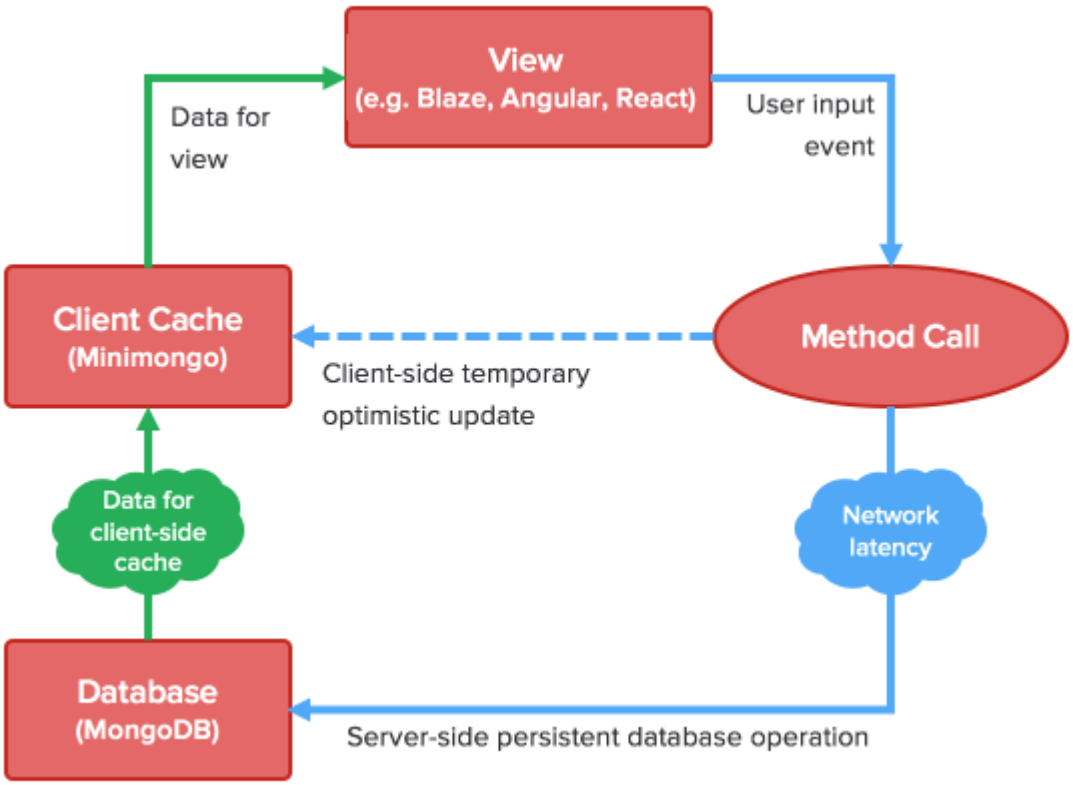
Meteor on sekä palvelin- että selainrajapinnat sisältävä JavaScript-pohjainen täyden pinon (full-stack) web-sovelluskehys, jonka avulla on mahdollista kehittää web- ja mobiilisovelluksia [48]. Sen kehittäjänä toimii Meteor Development Group (MDG). Meteor kuvataan reaaliaikaiseksi sovelluskehykseksi, jossa tehdyt muutokset näkyvät välittömästi selaimessa. Tästä välittömyydestä käytetään myös termiä reaktiivisuus. Meteor on rakennettu Node.js-pohjaisen palvelimen päälle, jolloin sekä palvelin että asiakaspuoli on toteutettavissa JavaScriptillä. Meteor on vienyt tämän ajatuksen jopa kooditasolle, sillä samassa tiedostossa voi olla asiakaspuolen ja palvelinpuolen koodia ja ne voidaan erottaa toisistaan käyttäen isClient- ja isServer-muuttujien arvoja. Meteor käyttää käyttöliittymän toteutuksessa omaa Blaze-sivupohjamoottoriaan, joka on suunniteltu hyödyntämään Meteorin reaktiivisuutta. Käyttöliittymä on mahdollista toteuttaa myös Angular- tai React-sovelluskehysten käyttämillä sivupohjilla. Kuvassa 13 on nähtävissä yleiskuva Meteor-sovelluskehyksen rakenteesta.



Kuva 13. *Meteorin rakenne [49].*

Tiedonvälityksessä käytetään Meteoria varten kehitettyä DDP-protokollaa (Distributed Data Protocol), joka perustuu Web Sockets -rajapinnan toteuttavaan SockJS-kirjastoon. Protokolla perustuu määrätyn muotoisten JSON-viestien välittämiseen asiakkaan ja palvelimen välillä. Tiedon synkronointi asiakkaan ja palvelimen välillä perustuu julkaise-tilaa-periaatteeseen, jossa asiakas kuuntelee Meteorin tietokantana käyttämän MongoDB:n operaatiolokissa tapahtuvia muutoksia. Tätä toiminnallisuutta kutsutaan nimellä Livequery. Jotta nämä JavaScript-tasolla muuttuneet arvot saadaan vielä välitettyä käyttöliittymään, päivittää Meteorin Tracker-komponentti muutokset DOMiin.

Meteor.js on suunniteltu reagoimaan nopeasti muutoksiin, vaikka verkon latenssi olisikin suuri, sillä muutos tehdään ensin selaimen lokaaliin JavaScript pohjaiseen Minimongo tietokantaan. Tästä mekaniikasta käytetään nimitystä optimistinen käyttöliittymä (Optimistic UI). Tämä tapa mahdollistaa käyttäjän itse tekemien muutosten näkymisen välittömästi, mutta muiden tekemät muutokset näkyvät vasta operaatiolokia seuraamalla. Tietokantaa voidaan myös käyttää ilman tätä latenssin kompensaatiota. Tätä kommunikointitapaa havainnollistetaan kuvassa 14.



Kuva 14. *Meteorin optimistinen käyttöliittymä [50].*

Käyttäjän tekemä muutos synnyttää siis kaksi päivitystä, joista toinen tehdään lokaaliin Minimongo-tietokantaan ja toinen lähetetään palvelimelle. Tiedon tallennuttua palvelimen MongoDB:hen verrataan siitä asiakkaalle siirtyvää arvoa lokaalissa Minimongossa olevaan arvoon ja korjataan se vastaamaan palvelimelta saatavaa arvoa, jos siihen on tarvetta.

4.2 D3.js

Sovelluksessa käytetyn D3.js-kirjaston avulla voidaan helposti muokata dokumentin HTML-, SVG- ja CSS-rakennetta. Keskeisenä suunnitteluperiaatteena on ollut DOMin CSS-tyylisen selector-funktion käyttö, jolla voidaan rajata tietty DOMin oliojoukko, jonka ominaisuuksia manipuloidaan kohdistamalla siihen ketjutettavia operaatioita jQueryn tapaan.

Tyypillisesti kirjastoa käytetään sivun graafisten esitysten DOM-rakenteen dynaamiseen muokkaamisen esityksiin liittyvän datan avulla, mistä D3 (Data Driven Documents) onkin saanut nimensä.

Kirjasto sisältää myös sovelluksen toteutuksen kannalta tärkeän transition-metodin, joka tekee siirtymän annettuun attribuutin arvoon määrätyssä ajassa käyttäen nykyisiä arvoja lähtötilanteena. Se on suunniteltu tarvittaessa keskeyttämään edellinen animaatio ja jatkamaan siitä uuden animaation alkaessa [51].

Kirjasto sisältää myös timer-funktion, joka sisäisesti hyödyntää, jos vain mahdollista, *requestAnimationFrame*-metodia ajoitukseen. Kirjaston tehokkuus perustuu siihen, että kaikki Timer-funktion kutsut, mukaan lukien transition-metodin generoivat timer-kutsut, sijoitetaan jonoon, jota ajetaan vain yhdessä yhteisessä *requestAnimationFrame*-takaisinkutsufunktiossa [52].

4.3 Haasteet teknologian soveltamisessa

Haasteena Meteorin käytössä, kuten muidenkin sovelluskehysten käytössä, on se, että asiat on tehtävä sovelluskehykselle ominaisella tavalla ja siitä poikkeaminen voi olla hyvin hankalaa. Meteorissa näitä pakollisia ominaisuuksia ovat esimerkiksi DDP-protokollan käyttäminen ja MongoDB–tietokannan käyttäminen. Dokumenttipohjaisilla tietokannoilla kuten MongoDB voi olla hankala tehdä monimutkaisia kyselyitä useamman taulun välillä ja ne voidaan joutua jakamaan useaan erilliseen kyselyyn [53]. Meteorille tosin löytyy muiden tietokantojen käytön mahdollistavia kolmannen osapuolen paketteja, joiden täydellistä toimintaa ei kuitenkaan voi taata. Mahdollisissa virhetilanteissa suuresta abstraktiotasosta voi olla haittaa, kun pitääkin perehtyä sovelluskehyksen kätkettyyn toteutukseen.

Koska Meteor on täyden pinon sovelluskehys, saattaa olla hankala sovittaa se toimimaan jo entuudestaan toteutetun palvelin- tai asiakaspään toteutuksen kanssa. Meteor käyttää omaa Atmosphere-paketointia, jossa on rajallinen määrä ainoastaan Meteoria varten kehitettyjä kolmannen osapuolen paketteja. Versiosta 1.3 lähtien lähtien Meteor tukee kuitenkin myös npm-paketteja, mutta sovellus oli toteutettu käyttäen Meteorin versiota 1.2.1 ja käytetyt ratkaisut perustuivat sen asettamiin rajoituksiin.

5. WEB-SOVELLUS

Luvussa kuvataan pääpiirteittäin tilannekuvasovellus ennen muutoksien tekemistä. Aliluvussa 5.1 käydään läpi sovellukselle esitetyt keskeiset toiminnalliset vaatimukset, aliluvussa 5.2 sovelluksen toteutus sekä aliluvussa 5.3 suorituskyvyssä havaitut puutteet, joita työssä yritetään korjata.

5.1 Sovelluksen vaatimukset

Keskeistä sovelluksessa on käyttäjien mahdollisuus liikuttaa omistamiaan objekteja niin, että kaikki käyttäjät näkevät reaaliaikaisesti ja samankaltaisesti objektien suhteellisen sijainnin ja kulkusuunnan. Yhtä aikaa näkyvien objektien maksimimäärä on noin 30, ja ne on pystyttävä identifioimaan tekstitunnisteella. Vaikka suuri kuvataajuus on käyttökokemukselle tärkeätä, tärkeintä tämän diplomityön sovelluksessa on kuitenkin eri käyttäjien näkemä samanaikainen ja yhdenmukainen tilannekuva.

Interaktiivisuudella on vaatimuksena, että käyttäjän syöte rekisteröidään siedettävässä ajassa. Vertailukohtana voidaan pitää käytettävyystutkimuksia, joissa noin 100 millisekunnin viiveellä käyttäjä kokee syötteeseen reagoitavan välittömästi. Jos vaste on yli sekunnin, käyttäjä huomaa viiveen, mutta käyttäjän huomio pysyy silti sovelluksessa. Noin kymmenen sekunnin ja sen yli menevällä viiveellä käyttäjän huomio herpaantuu, ja tarvitaan jokin visuaalinen palaute ajasta, joka reagointiin vielä kuluu [54].

Sovelluksen halutaan olevan käytettävä mahdollisimman monella laitetyypillä huomioiden kuitenkin, että vanhempia laitteita jatkuvasti putoaa pois käytöstä. Käyttöliittymän on oltava käytettävä niin hiirellä kuin kosketusnäytölläkin.

Sovelluksen on toimittava ilman asennettavia selainliitännäisiä, sillä niiden asentaminen voi mobiililaitteiden kohdalla olla hankalaa tai mahdotonta. Lisäksi on huolehdittava, että sovellus toimii eri selaimilla samalla tavalla.

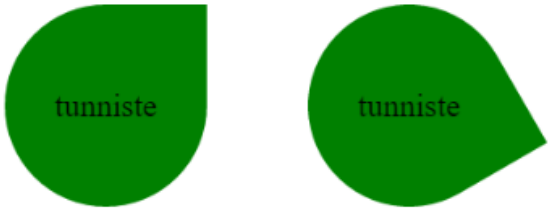
Sovelluksen odotetaan toimivan ainakin WiFi-verkossa, mutta suotavaa olisi, että sitä voisi käyttää myös 4G- tai jopa 3G-verkossa.

5.2 Sovelluksen toteutus

Alkutilanteessa tilannekuvasovelluksen objektien sijaintien päivittäminen oli hoidettu seuraavasti: Käyttäjän alkaessa raahata objektia käynnistetään ajastin, joka vastaa haluttua päivitysintervallia. Tämän ajastimen päätyttyä lasketaan piste, johon objekti olisi kulkenut vastaavassa ajassa korkeintaan tietyllä maksiminopeudella liikutettavaan suuntaan. Tämä piste tallennetaan sitten tietokantaan. Asiakassovelluksen saadessa tiedon objektin

sijainnin muutoksesta asetetaan D3.js-kirjaston transition-metodilla objektin sijainnille uusi animointipäämäärä havaittuun pisteeseen. Animoinnin ajalliseksi kestoksi asetetaan hieman pidempi kuin sijainnin päivitysintervalli. Uuden päivityksen saapuessa edellinen siirtymä keskeytetään ja aletaan siirtyä kohti uutta pistettä, jolloin liike saadaan näyttämään jatkuvalta.

Kuvassa 15 on nähtävissä sovelluksessa käytettävän SVG-objektin kaltainen objekti, joka koostui kahdesta päällekkäin olevasta elementistä: objektit toisistaan erottavasta tunnisteesta ja kuviosta, jossa menosuunta on esitetty nokalla. Liikutettaessa objektia vaihdetaan sen x- ja y-attribuutteja molemmille elementeille ja pyörittäessä asetetaan transform-attribuutin rotate-arvo vain kuvioelementille, jolloin tunniste pysyy paikallaan vaakatasossa.



Kuva 15. *Sovelluksessa käytettävän SVG-objektin kaltainen kuvio.*

Toteutuksen järkevyyttä voidaan arvioida vertaamalla sitä yleisesti peleissä käytettyihin tekniikoihin. Toteutus eroaa peleissä käytettävistä tekniikoista siinä, että se ei sisällä hallittua pelisilmukkaa, jossa suoritetaan jatkuvasti peräkkäin syötteiden käsittelyä, tilan päivittämistä ja grafiikan piirtämistä. Animointeja toteutettaessa on tärkeää, että niihin kuluva aika on riippumaton kuvataajuudesta, koska eri laitteet ja selaimet suorittavat ohjelmaa eri nopeuksilla. D3.js:n transition-metodi suoriutui tästä ongelmakohdasta hyvin.

Jotta verkkoviiveen vaikutus saataisiin minimoitua, peleissä käytetään usein asiakaspuolen ennakointia, jossa objektia liikutetaan tavalla tai toisella asiakaspuolella ennen kuin viesti uudesta sijainnista on saapunut palvelimelle [55]. Meteorin käyttämä Optimistic UI -tekniikka perustuu myös tähän ajatukseen, mutta sillä erolla, että muutos on nähtävissä etukäteen vain tiedon lähettäjällä. Muut joutuvat edelleen odottamaan palvelimelta saapuvaa viestiä. Tämä johtaisi eriaikaisiin tilannekuviin eri näytöillä, joten Optimistic UI -tekniikkaa ei hyödynnetty sovelluksessa. Ratkaisu, jossa sekä lähettäjä että vastaanottajat näkisivät liikkeen ennakoidusti extrapoloimalla objektin aiemmista sijainneista ja niihin pohjautuvista laskennallisista nopeuksista, olisi herkkä suunnan muutoksille. Siksi päädyttiin interpoloimaan D3.js:n transition-metodin avulla objektin edellisen ja uuden tilan välistä siirtymää. Tässä ratkaisussa raahaamisen aiheuttaman liike on pehmeää, mutta raahaamiseen reagoiminen on hieman hitaampaa. Kuitenkin vielä puolen sekunnin viivekin siirtymisessä raahaamisen osoittamaan paikkaan koettiin siedettävänä.

5.3 Havainnot suorituskyvystä

Suorituskykytestauksissa pyrittiin mallintamaan sovelluksen suorituskyvyn kannalta pahinta mahdollista tilannetta, jossa kaikkia tilannekuvan objekteja liikutetaan samanaikaisesti ja kunkin uudet koordinaatit lähetetään palvelimelle ja muille osallistujille yksittäisinä viesteinä. Kokeessa havainnoitiin, pysyvätkö kaikkien eri laitteiden näkyvät yhtenäisinä.

Testit suoritettiin aluksi lähiverkossa, sillä jos jo siinä ilmenee ongelmia, verkkoviiveen kasvaessa tilanne vain tulee pahentumaan.

Sovelluksen testauksessa käytettävät laitteet on listattu taulukossa 1. Valituilla laitteilla ja selaimilla saatiin hyvin katettua suosituimmat selaimet ja käyttöjärjestelmät ja nähtyä, onko niiden välillä suuria eroja. Pelkästään selainten väliseksi vertailuksi testaus ei kuitenkaan jäänyt, sillä laitteiden komponentit olivat hyvin eri tehoisia.

Taulukko 1. *Testauksessa käytetyt laitteet.*

Käytetty laite	**Laitteen Käyttämä selain**
Samsung Galaxy Tab 3 7.0 -tabletti	Chrome
Nokia Lumia 830 -puhelin	Internet Explorer
Apple Ipad Air -tabletti	Safari
Lenovo ThinkPad G580 -kannettava	Chrome

Testit suoritettiin käyttäen yhtäaikaisesti 10 liikuteltavaa objektia 150 millisekunnin paikkatiedon päivitysintervallilla. Testejä jatkettiin nostamalla liikuteltavien objektien määrää sekä muuttaen päivitysintervallia niin suuremmaksi kuin pienemmäksi.

Suorituskykytesteissä havaittiin, että välitettäessä paljon paikkaviestejä lyhyessä ajassa Meteorin käyttämä Livequery kävi hitaaksi. Lisäksi sijaintien lähetyksen päätyttyä objektit kirivät päätepisteeseen, joten samanaikaisesti lähettäessä ja vastaanottaessa on ongelmia. Meteorin tapa välittää tietoa tallentamalla se ensin tietokantaan ja kuuntelemalla sen muutoksia on hidas, jolloin viestit puskuroituvat.

Lisättäessä liikuteltavien objektien määrää oli havaittavissa, että liike hidastui kaikilla laitteilla yhä enemmän. Tätä voitiin kompensoida lähettämällä paikkaviestejä harvemmin, mikä näkyi toisaalta myös kasvavana viiveenä käyttäjän syötteisiin reagoimisessa.

Lisäksi testeissä käytetty Windows Phone, Lumia 830, jäi jälkeen viestien käsittelyssä verrattuna muihin selaimiin. Sen tilannekuva pysyi samana vain tilanteissa, joissa kaikki objektit eivät olleet liikkeessä tai niiden päivitysintervalli oli kasvatettu niin pitkäksi, ettei järjestelmä enää reagoinut riittävän nopeasti käyttäjän syötteisiin.

Testeissä havaittiin myös, että piirrettävää objektia yksinkertaistamalla sen piirtäminen saatiin nopeammaksi ja tilannekuva pysyi tahdissa myös tilanteissa, joissa se aikaisemmin jäi jälkeen. Erityisen raskaaksi suorituskyvyn kannalta osoittautui objektien tunnisteena toimivan SVG-tekstin piirtäminen; sen käyttö oli kuitenkin välttämätöntä, jotta muuten samanlaiset objektit voidaan erottaa toisistaan.

6. SUORITUSKYVYN PARANTAMINEN

Suorituskyvyn parantamiseksi itse ohjelmaan tehtiin toimenpiteitä, jotka on kuvattu aliluvussa 6.1. Siinä on myös esitelty mittaustuloksia, jotka kuvaavat muutosten vaikutusta. Aliluvussa 6.2 arvioidaan vielä erikseen subjektiivisesti toiminnallisuuden tilaa muutosten jälkeen.

Sovelluksen jatkokehitystä varten sen SVG-pohjaista esitystapaa verrattiin muihin vaihtoehtoisiin renderöintitapoihin, muun muassa canvas- ja WebGL-pohjaisiin ratkaisuihin. Testit tehtiin itse ohjelman ulkopuolella simuloiden itse ohjelman piirtotilannetta. Testit ja niiden tulokset on kuvattu aliluvussa 6.3.

6.1 Havaittujen ongelmien korjaaminen

Ratkaisuksi Meteorin kommunikoinnin tehostamiseksi käytettiin Meteorin Atmosphere-pakettien hallintasivulta löytynyttä Streamy-kirjastoa [56], joka mahdollisti DDP:n käyttämän SockJS-kirjaston hyödyntämisen asiakas-palvelin-kommunikoinnissa. Tällöin saatiin ohitettua Livequeryssä olennainen tietokantaan tallentaminen ja tietokannan muutosten tarkkaileminen. Streamyn avulla viestit saatiin välitettyä osallistujalta suoraan palvelimen kautta takaisin lähettäjälle ja toisille osallistujille.

Yllä olevien toteutustekniikoiden nopeutta vertailtiin testissä, jossa asiakas toistuvasti lähetti palvelimelle uuden viestin saadessaan palvelimelta vastauksen edelliseen viestiin. Testissä käytetty viesti vastasi sovelluksessa käytettyä JSON-tyyppistä viestiä, jossa viesti sisälsi objektin sijainnin x- ja y-koordinaatteina, tunnisteen ja suunnan. Testissä suunta-attribuutin arvo korvattiin laskurilla, jota kasvatettiin palvelimella aina viestin saapuessa. Lähetystä ja vastaanottoa toistettiin 1000 kertaa. Mittaamalla aikaa ensimmäisen lähetetyn ja viimeisen vastaanotetun viestin välillä saatiin taulukon 2 mukaiset tulokset testatuilla laitteilla. Aika mitattiin viisi kertaa ja taulukossa ilmoitettu arvo on keskiarvo näistä mittauksista. Testit suoritettiin lähiverkossa, ja testilaitteina käytettiin aikaisemmin mainittuja mobiililaitteita Lenovo G580 -kannettavan toimiessa palvelimena. Tuloksia ei voida pitää täysin absoluuttisina vaan lähinnä havainnollistavina johtuen siitä, että laitteilla voi olla taustaprosesseja päällä, niiden havaitsema verkon voimakkuus voi vaihdella ja verkon kuormitus eri testien välillä saattoi vaihdella.

Taulukko 2. *1000 viestin lähettämiseen kulunut aika (ms) per viesti eri kommunikointitavoilla.*

Käytetty laite	**Meteor**	**Streamy**
Samsung Galaxy Tab 3 7.0	16,8	7,0
Nokia Lumia 830	17,3	6,1
Apple Ipad Air	16,5	5,6

Tuloksista voidaan nähdä, että viestin kulkuaika pienenee olennaisesti siirryttäessä käyttämään Streamyä kommunikoinnissa.

Streamyyn siirtymisen jälkeen havaittiin Windows Phonen edelleen jäävän jälkeen reaaliaikaisuudessa, kuten se oli jäänyt jälkeen alkuperäisessäkin toteutuksessa.

Streamyn avulla päästiin paremmin käsiksi viestien välitysmekanismiin: tällöin oli mahdollista kokeilla toteutustekniikkaa, jossa palvelin kerää asiakassovelluksilta saatuja uusia paikkakoordinaatteja, ja lähettää ne yhtenä koosteviestinä tietyn väliajoin takaisin asiakassovelluksille. Viestin koko kasvoi, mutta määrä väheni olennaisesti. Koosteviestin päivitystaajuus asetettiin noin puoleen asiakkaan lähettämien viestien päivitystaajuudesta.

Yksittäisten viestien välitysnopeutta verrattuna koosteviestin välitysnopeuteen testattiin Streamyssa lähettämällä edestakaisin – kuten edellisessä kokeessa – tuhat kertaa yksittäinen viesti ja 20 yksittäisestä viestistä muodostettu koosteviesti. Testiä toteuttaessa huomattiin, että koostettuun viestiin tuli paljon rakenteeltaan identtisiä JSON-objekteja, joiden nimi-arvo-parien nimet ja lukumäärä pysyivät muuttumattomina. Tällöin objekteilta voitiin poistaa niiden nimet sekä tunnisteattribuutti kokonaan. Tällöin viestit koostuivat vain JSON-taulukosta, jossa on numeroita: eri objektien arvot ovat pääteltävissä koosteviestin indeksin avulla. Tällöin välitettävän viestin koko pieneni olennaisesti. Tämän testin tulokset on listattu taulukkoon 3.

Taulukko 3. *1000 viestin lähettämiseen kulunut aika(ms) per viesti kahdella eri koostamistavalla.*

Käytetty laite	**20 viestiä koostettu yhteen**	**20 viestiä koostettu yhteen tunnisteet poistettuina**
Samsung Galaxy Tab 3 7.0	8,4	7,6
Nokia Lumia 830	9,8	6,9
Apple iPad Air	7,6	6,3

Taulukoita 2 ja 3 vertailemalla voidaan havaita, että vaikka tietoa välitettiin koosteviestissä 20 kertaa enemmän kuin yhdessä viestissä, ei viestien välittämiseen kuluva aika kasvanut läheskään samassa suhteessa. Karsittaessa viestit kompaktiin muotoon, päästiin jopa hyvin lähelle alkuperäisen yksittäisen viestin välittämiseen kuluvaa aikaa. Sovelluksesta mitatut ajat olisivat vielä lähempänä taulukon 2 yksittäisen viestin välittämisaikaa, sillä testeissä koosteviestiä lähetettiin edestakaisin asiakkaan ja palvelimen välillä, kun taas sovelluksessa yksittäinen asiakas lähettää palvelimelle vain oman sijaintinsa sisältävän viestin.

Käytännössä käyttäen Streamyä ja koosteviestin kompressointia liike ei enää hidastunut yhtä alhaisilla päivitystaajuuksilla kuin MongoDB:tä käyttävässä versiossa. Siirtyminen viestien koostamiseen pidensi kuitenkin hieman kommunikaatioviivettä ja heikensi yksittäisen objektin liikuttelun sulavuutta; järjestelyllä kuitenkin saavutettiin se, että objektien määrän kasvaessa mikään laite ei ala jäädä yhä pahemmin jälkeen reaaliaikaisuudesta.

Yritettäessä pienentää koosteviestien lähetysintervallia testatusta 150 millisekunnin intervallista Lumia 830 oli ensimmäinen laite, jolla viestit kumuloituivat asiakkaalle johtaen yhä kasvavaan viiveeseen.

6.2 Suorituskyvyn arviointi korjausten jälkeen

Aliluvussa 6.1 mainittujen korjausten jälkeen suorituskykyä saatiin parannettua vaatimusten mukaisiksi: kaikilla testattavilla laitteilla ja osallistujilla näkyisi sama tilanne. Samalla kuitenkin luovuttiin yksittäisten viestien lähettämisestä, jolloin reaaliaikaisuus kärsi. Alkuperäiseen tavoitteeseen siis päästiin, mutta suorituskyky jätti silti toivomiseen varaa.

Streamyn käyttöönottaminen ei ratkaissut ongelmaa, jossa yhdenkin viestin viivästyminen jumittaa koko sovelluksen suorituksen. Streamyn käyttöönottaminen asetti myös haasteita istunnonhallinnalle, sillä Livequerystä luovuttaessa ei istuntokohtaista dataa

pystytty enää istuntoon kuuluville käyttäjille, koska tietokannan istuntokohtainen informaatio menetettiin. Ratkaisuksi ongelmaan viestejä välittävään palvelimeen räätälöitiin toiminto, joka pitää kirjaa kuhunkin istuntoon liittyvistä WebSocket-yhteyksistä ja välittää viestit vain niiden välillä.

Testeistä kävi myös selväksi, että kaikki selaimessa tapahtuva toiminta vaikuttaa toisiinsa ja suorituskykyyn selaimen JavaScriptin yksisäikeisyyden takia. Onkin tärkeää, että kaikkiin operaatioihin kuluu mahdollisimman vähän aikaa, jotta ne eivät blokkaa toisia operaatioita. Galaxy Tab 3 ja Chrome-selain osasi vuorotella piirtämistä ja viestien välitystä paremmin kuin Lumia 830, sillä vaikka sen kuvataajuus laski selvästi, pysyi se silti viestienvälityksessä mukana. Lumia 830:lla oli taas havaittavissa, että palvelimelta saatavat viestit puskuroituvat ja laitteen Internet Explorer -selain jää käsittelemään niitä jonosta: laite ei pysy tahdissa, sillä suurin osa sen prosessoriajasta kuluu grafiikan piirtämiseen.

6.3 Eri piirtotekniikoiden suorituskyvyn vertailua

Koska havaittiin, että myös grafiikan piirtämisen nopeudella on verkkoliikenteen lisäksi tärkeä rooli laitteiden pysymisessä samassa tahdissa, suorituskyvyn parantamiseksi vertailtiin eri tapoja piirtää grafiikkaa ja sitä, eroavatko niitä käytettäessä saavutetut kuvataajuudet merkittävästi toisistaan. Jos löydettäisiin keino, jolla grafiikan piirtäminen sujuisi paljon nopeammin, auttaisi se myös asiakas-palvelin-kommunikaatiossa, koska viestijonon käsittelyyn jäisi enemmän aikaa.

Testit suoritettiin toteutettavan sovelluksen ulkopuolella, sillä niiden liittäminen muun toteutuksen kanssa yhteensopivaksi ei ollut yksinkertaista. Lisäksi D3.js-kirjaston transition-metodin aiheuttaman animaation kuvataajuuden mittaaminen ei onnistu, koska sen kuvataajuutta ei voi itse kontrolloida. Niinpä testit toteutettiin liikuttamalla vaakatasossa ja pyörittämällä *requestAnimationFrame*-silmukassa 400 kertaa vaihtelevaa määrää tilannekuvan mukaisia objekteja muuttaen joka kuvassa objektin sijaintia x-akselilla ja objektin rotaatiota. Kun tiedetään piirrettävien kuvien määrä, voidaan kuvataajuus laskea mittaamalla aika ensimmäisestä kuvan piirtämisestä viimeiseen kuvan piirtämiseen kulunut aika. Testeissä keskityttiin kolmeen aikaisemmissa mittauksista käytettyihin laitteisiin Lumia 830, Galaxy Tab 3 ja iPad Air.

Sovelluksessa objektien siirrossa käytetään paikkakoordinaattien vaihtamiseen SVG:n x- ja y-attribuutteja. Tätä toteutusta testeissä verrattiin SVG:n transform -attribuutin translate-määritteen avulla toteutettuun siirtoon. Kummassakin tapauksessa objektin pyörittäminen toteutettiin SVG:n transform-attribuutin rotate-määritteellä. Vertailuun otettiin mukaan myös CSS-animaatiot, vaikka niiden toimimattomuus SVG:n sisäkkäisten elementtien kanssa Internet Explorer- ja Edge-selaimilla tiedostettiinkin. Näin tekemällä saatiin tietoa kuitenkin niiden CSS-animaatioiden suorituskyvystä. Myöskin SVG:n transform-attribuutin translate-määritteen kanssa oli ongelmia Edge-selaimella. Suorituskykytesteissä testattiin myös HTML-elementtien siirtämistä käyttäen pelkkiä CSS-

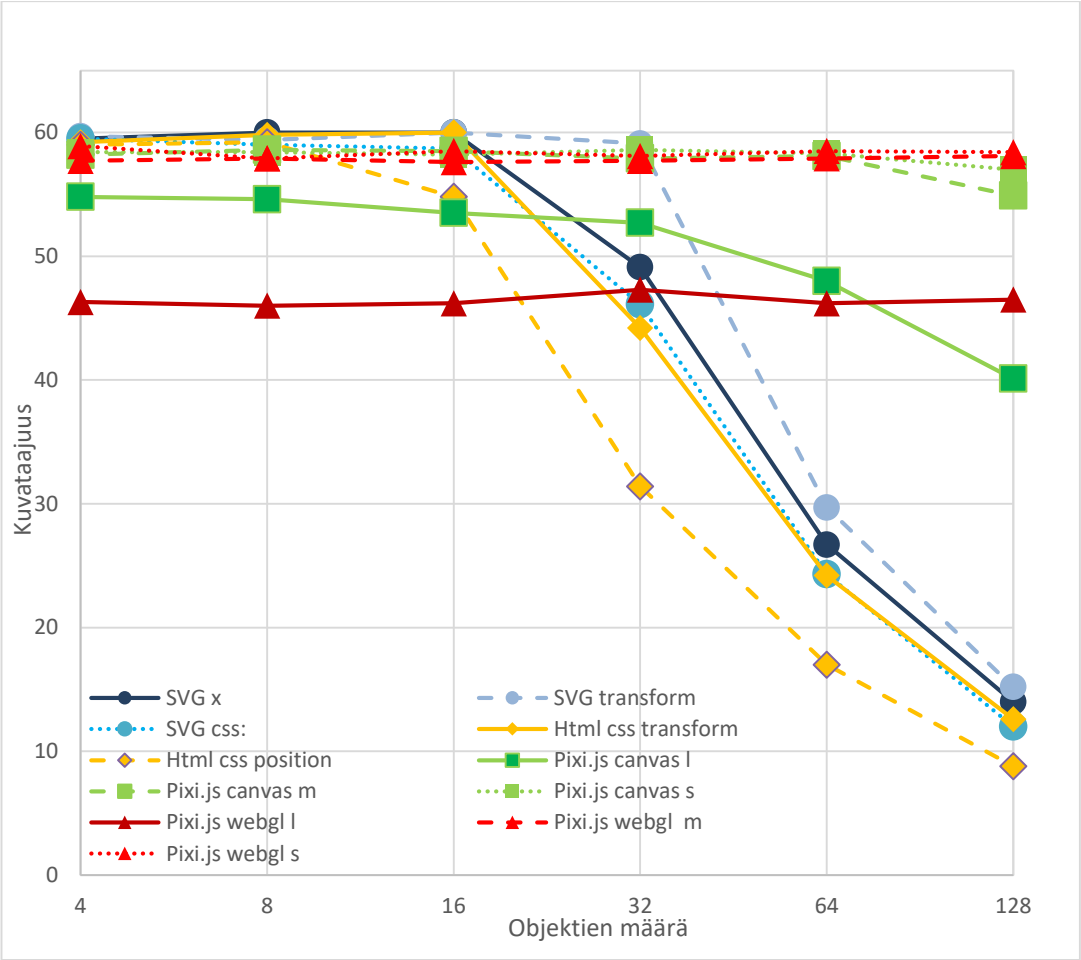
animaatioita vaihtamalla HTML-elementtien top- ja left-tyyliominaisuuksia. CSS-animaatiotestissä havaittiin, että kohdistamalla *will-change*-tyylisääntö objektin transform-ominaisuuteen saatiin parannuksia kuvataajuuteen. Kun sama tehtiin left-tyyliominaisuutta vaihtaessa HTML-elementtien kanssa, kuvataajuus parani iPadilla, mutta laski Galaxy Tab 3:lla, joten muutos jätettiin vain ensimmäiseen testiin. WebGL- ja canvas-pohjaiset renderöintitestit toteutettiin Pixi.js-kirjaston versiolla 3 käyttämässä kirjaston WebGLRenderer- ja CanvasRenderer-funktioita. Pixi.js-kirjastoa käytettäessä liikuteltavat objektit olivat kaksiulotteisia bittikarttoja eli spritejä. Sprite-objektien liikuttaminen tapahtui muuttamalla niiden position.x ja rotation arvoja.

SVG-pohjaisissa testeissä objektit luotiin kuten sovelluksessa. Piirtoalustana toimivan SVG:n lapsielementeiksi luotiin SVG:n ryhmälementtejä (g-elementtejä), jotka koostuivat neliöstä ja numerotunnisteesta käyttäen SVG:n rect- ja text-elementtejä. HTML-pohjaisissa testeissä objektit luotiin sijoittamalla div-elementtien sisään kaksi span-elementtiä, joista toinen määritettiin CSS-tyyleillä neliöksi ja toinen toimi numerotunnisteena. Pixi.js-pohjaisissa testeissä grafiikka koostui kahdesta spritestä, jotka vastasivat sovelluksessa käytettävää kuviota ja tunnistetta.

Testejä suunniteltaessa havaittiin, että piirron suorituskyvyssä voidaan saavuttaa parempi kuvataajuus viittaamalla objekteihin valmiiksi tallennetuilla viitteillä sen sijaan, että ne etsitään joka kuvaa renderöitäessä erikseen DOMista. Varsinaisessa sovelluksessa tällä tavalla ei kuitenkaan yhtä suuria parannuksia tultaisi näkemään, sillä objekteihin kohdistuvat DOM-kyselyt tapahtuvat vain uuden paikkatiedon sisältävää viestiä käsiteltäessä. Taulukossa 4 on kuvattu ensin kuvaajissa käytettävien testien lyhenteet. Kuvissa 16,17 ja 18 on esitetty testeissä mitatut kuvataajuudet objektimäärän funktiona Lumia 830-, Galaxy Tab 3- ja iPad Air -laitteilla.

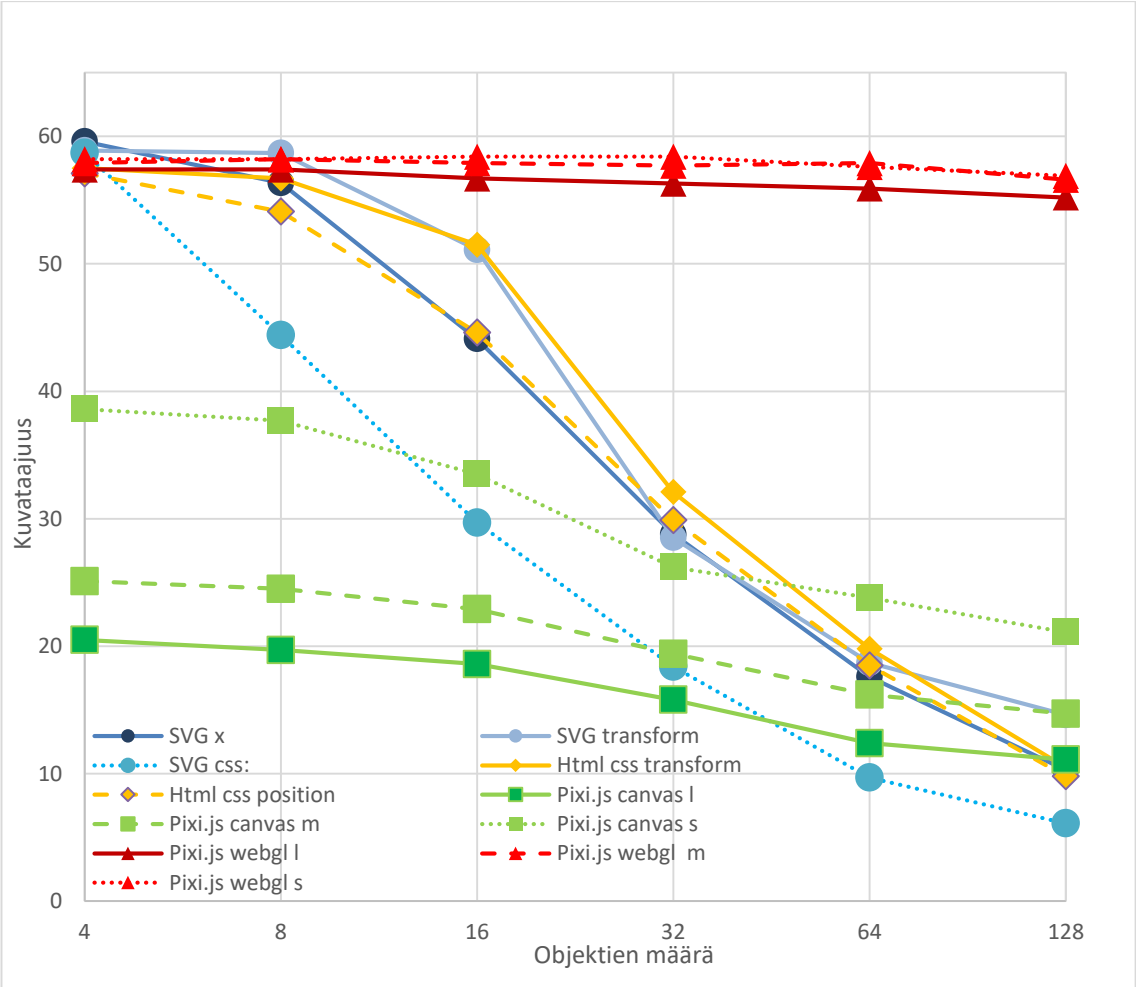
Taulukko 4. *Suorituskykytestien kuvaajissa käytetyt lyhenteet.*

Käytetty lyhenne	**Kuvaus testistä**
SVG x	SVG-objekteja siirretään muuttamalla x-attribuutin arvoja ja pyöritetään käyttämällä transform-attribuutin rotate-määritettä
SVG transform	SVG-objekteja siirretään ja pyöritetään muuttamalla transform-attribuutin translate- ja rotate-määritteitä
SVG css	SVG-objekteja siirretään ja pyöritetään käyttämällä CSS:n transform-tyyliominaisuuden translate- ja rotate-määritteitä
HTML css transform	HTML elementtäjä siirretään ja pyöritetään käyttämällä CSS:n transform-tyyliominaisuuden translate- ja rotate-määritteitä
HTML css position	HTML elementtäjä siirretään muuttamalla niiden left-tyyliominaisuutta ja pyöritetään käyttämällä CSS:n translate-ominaisuuden rotate-määritettä
Pixi.js webgl l, m ja s	WebGL testi käyttäen Pixi.js WebGLRenderer-funktiota. Sprite-objektien siirtäminen ja pyörittäminen on toteutettu muuttamalla position.x ja rotate-arvoja. Testit on toistettu käyttäen resoluutioita 1920x1080(l), 960x540(m) ja 640x360(s)
Pixi.js canvas l, m ja s	Canvas-testi käyttäen Pixi.js CanvasRenderer-funktiota. Sprite-Objektien siirtäminen ja pyörittäminen on toteutettu muuttamalla position.x ja rotate-arvoja. Testit on toistettu käyttäen resoluutioita 1920x1080(l), 960x540(m) ja 640x360(s)



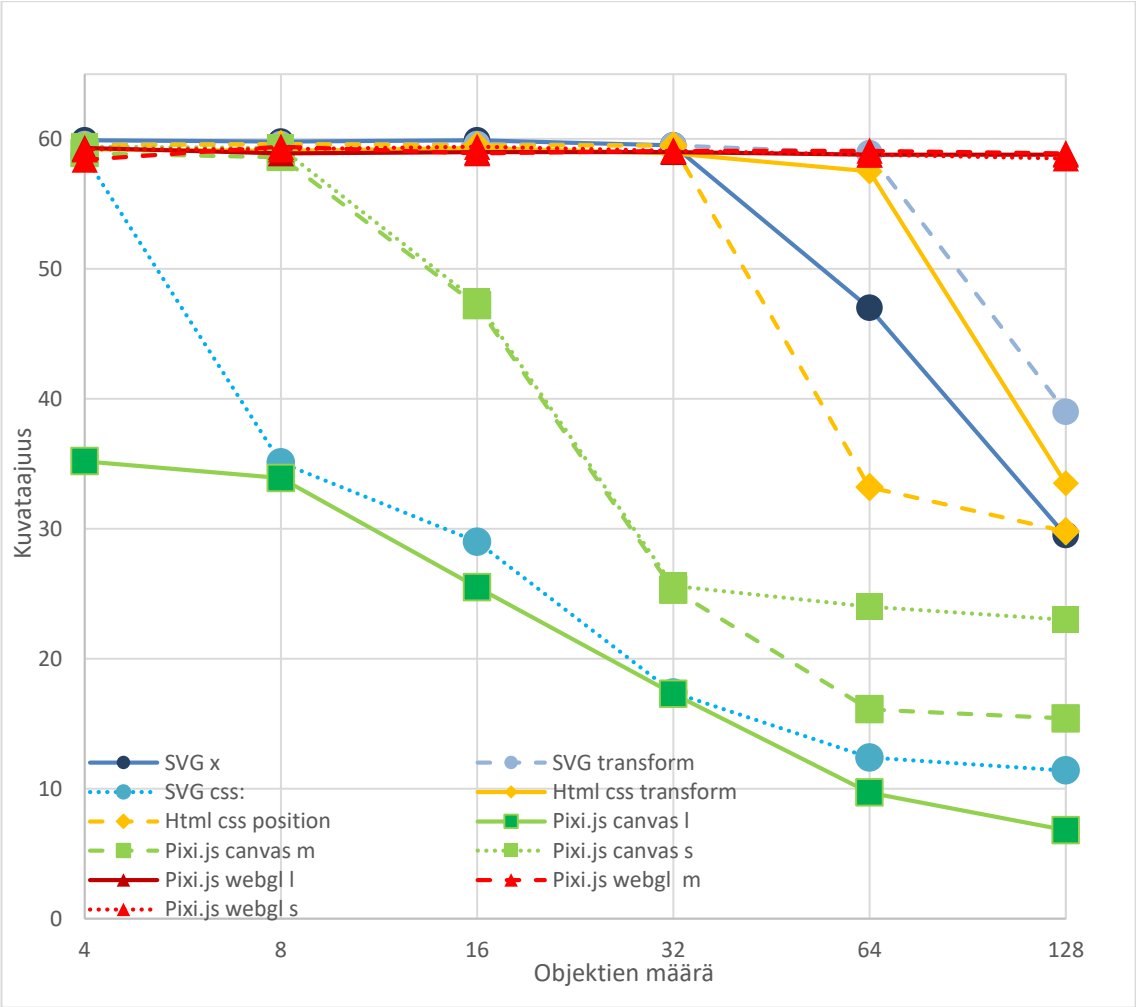
Kuva 16. *Lumia 830:n kuvataajuus objektien määrän funktiona eri piirtotekniikoilla.*

Lumia 830:n kohdalla paras tulos saavutetaan käyttämällä WebGL-pohjaista grafiikkaa. Suurimman resoluution kohdalla kuvataajuus on alhaisempi, mutta kyseinen resoluutio on merkittävästi suurempi kuin Lumia 830:n oman resoluutio. Tällöin tulos ei vastaa todellista käyttötilannetta. Huomattavaa oli Pixi.js-kirjaston CanvasRendererin hyvä suorituskyky samalla resoluutiolla. Tämä on osittain selitettävissä sillä, että Internet Explorer käyttää canvas-elementin kanssa suoraan laitteistokiihdytystä [57].



Kuva 17. *Galaxy Tab 3:n kuvataajuus objektien määrän funktiona eri piirtotekniikoilla.*

Galaxy Tab 3:n kohdalla WebGL:n paremmuus muihin piirtotekniikoihin nähden on selvästi huomattavissa objektimäärän kasvaessa. CanvasRenderer-pohjainen ratkaisu ei toimi yhtä hyvin kuin se toimi Lumian kanssa, sillä Chrome-selaimella ei ole vastaava laitteistokiihdytystä kuin Internet Explorer-selaimella.



Kuva 18. *iPad Airin kuvataajuus objektien määrän funktiona eri piirtotekniikoilla.*

iPad Airin kohdalla on nähtävissä jälleen WebGL:n paremmuus muihin piirtotekniikoihin verrattuna, mutta laitteen tehokkuudesta johtuen myös muut piirtotekniikat pärjäävät isojen objektimäärien kanssa. Pixi.js:n CanvasRenderer ei tosin suoriutunut hyvin kuin aivan pienillä objektimäärillä.

Testeistä havaitaan, että ainoastaan WebGL-pohjainen grafiikka onnistui pitämään hyvän kuvataajuuden objektien määrän kasvaessa kaikilla laitteilla. SVG-pohjaista grafiikkaa käytettäessä DOM-puun uudelleenrakentaminen on raskasta. Myöskään CSS-animaatiot eivät suoriutuneet testissä hyvin. Pixi.js-kirjasto on myös suunniteltu erityisesti WebGL:ää varten, joten sen 2D-canvas-toteutus ei välttämättä ole paras mahdollinen olemassa olevista kirjastoista. Laitteiden välillä on huomattavissa laitteiden tehoeron lisäksi se, että eri piirtotekniikoiden nopeus riippuu myös käytettävästä selaimesta ja käyttöjärjestelmästä. Suorituskyky liikuttamalla SVG-objekteja CSS-tyyliä muuttamalla vaihtelee merkittävästi eri laite-selain-yhdistelmien kesken. Testeistä on myös nähtävissä, kuinka objektien määrän kasvaessa DOMin muokkaukseen perustuvien ratkaisujen kuvataajuus laskee nopeammin kuin canvas- ja WebGL-pohjaisilla ratkaisuilla.

6.4 Jatkokehitysajatukset

Suorituskyvyssä jäi edelleen parannettavaa. Sitä voitaisiin parantaa tarkastelemalla kehittäjätyökalujen timeline-aikajanoja. Jos grafiikan piirtämisessä halutaan pysyä SVG:ssä, olisi tätä kautta saavutettavissa pieniä parannuksia optimoimalla, kuten pyrkimällä välttämään sivun uudelleen asemointia. SVG-objektin osana on myös SVG:llä piirretty tekstitunniste, joka voitaisiin piirtää myös tavallisena HTML-tekstinä.

Hieman työläämpänä ratkaisuna olisi suunnitella sovellus alusta asti sillä ajatuksella, että kyseessä on peli ja siinä käytettäisiin peleihin soveltuvia tekniikoita. Sovellus toteutettaisiin hyödyntäen WebGL-pohjaista grafiikkaa ja sen teossa voitaisiin käyttää jotain valmista WebGL:n käyttöä helpottavaa kirjastoa, kuten Pixi.js. Yksi tutkittava asia olisi verrata pelisilmukan etuja verrattuna käytössä olleeseen tilannetietopäivityksen käynnistämään sykliin.

Jos grafiikan piirtoa saataisiin kevyemmäksi, saattaisi olla mahdollista siirtyä takaisin käyttämään yksittäisiä sijaintiviestejä koosteviestien sijaan, jolloin liikkeen sulavuus voisi parantua. Liike voitaisiin saada reagoimaan muutoksiin nopeammin myös, jos sen toteuttamisessa sovellettaisiin peleissä käytettyä asiakaspuolen ennakointia. Tämä kuitenkin aiheuttaisi sen, että käyttäjät näkisivät oman objektinsa liikkeen muiden edellä.

Koska sovellus ei ole toteutettu käyttäen peleissä käytettyä pelisilmukkaa, uudet paikkaviestit saattavat turhaan rasittaa jo meneillään olevaa piirtoa. Pelisilmukkaa vastaava *requestAnimationFrame* on käytössä vain d3.js kirjaston transition-metodin sisäisessä toteutuksessa, mistä seuraa, että piirron viedessä kaiken prosessoriajan ei viestien vastaanottamiselle jää aikaa vaan ne puskuroituvat.

Käytettäessä canvasta piirtoalustana, voidaan sille piirrettävien objektien ominaisuuksia päivittää taustalla olevaan malliin aiheuttamatta välitöntä uudelleenpiirtämistä. SVG:tä tai muuta DOMiin vaikuttavaa tekniikkaa käyttäessä elementin attribuutin muuttaminen aiheuttaa sen uudelleenpiirtämisen. Selain osaa kuitenkin yleensä jäädä odottamaan nopeasti peräkkäin tapahtuvia DOM- tai SVG-rakenteen muutoksia ja toteuttaa muutokset yhtenä kokonaisuutena.

Tulevaisuudessa vanhat laitteet poistuvat uusien tehokkaampien laitteiden tieltä, jolloin grafiikan piirrossa tapahtuvat ongelmat eivät ole yhtä suuria. Käytössä oleva SVG-pohjainen grafiikka on kuitenkin selaimille raskasta verrattuna WebGL-pohjaiseen grafiikkaan, jonka käyttäminen mahdollistaisi suuremman määrän samanaikaisesti liikkuvia objekteja ilman, että kuvataajuus laskisi. SVG:stä luopuessa jouduttaisiin kuitenkin luopumaan skaalautuvuudesta, jonka avulla objektit näyttävät yhtä tarkoilta eri näyttöko'oilla ja resoluutioilla.

Verkkoliikenteestä voidaan saada tehokkaampaa WebRTC-tuen yleistyessä eri selaimissa ja sen käytöstä tullessa enemmän kokemuksia; se mahdollistaisi UDP-protokollaan perustuvan viestien välityksen, jolloin hukkuneita paketteja ei jouduttaisi lähettämään uudestaan vaan voitaisiin keskittyä uudempiin paketteihin.

Yksi suorituskykyä rajoittava tekijä oli myös käytetty Meteor-versio. Toteutettavan sovelluksen käyttämän version 1.2.1 mukana tulevan MongoDB:n versio 2.6.7 on hitaampi kuin 3.0 ja sitä uudemmat versiot [58]. Päivittäminen uudempaan versioon ei ole aivan suoraviivaista, sillä uudemmat versiot vaativat muutoksia ohjelmakoodiin [59; 60]. Version 1.4 myötä Meteor käyttää MongoDB:n uudempaa 3.2 versiota, jonka vaikutus MongoDB:n operaatiolokin seurantaan pohjautuvaan tiedonvälitykseen voitaisiin tutkia. Tieto välitettäisiin kuitenkin edelleen WebSocket-yhteyden avulla, joten Meteorin käyttämä kommunikointi ei tule ohittamaan nopeudessa puhdasta WebSocket-ratkaisua.

Eräänä jatkokehitysideana voisi myös tutkia sovelluksen toteuttamista ilman Meteor-sovelluskehystä, jolloin ohjelma ei sisältäisi ylimääräistä ohjelmistokehykseen liittyvää koodia. Tällöin esimerkiksi WebSocket-kirjastoista voitaisiin koittaa Socket.IO:n hyödyntämistä.

Vaikka natiivisovelluksena toteutettuna sovellus voisi hyödyntää UDP-protokollaa, monisäikeisyyttä sekä tehokkaampaa laitteistokiihdytystä, ei natiivisovellus välttämättä kuitenkaan olisi hyvä idea. Eri alustoille olisi ohjelmoitava omat asiakassovelluksensa ja huolehdittava, että ne käyttäytyisivät samankaltaisesti ja kommunikoisivat toistensa tai palvelimen kanssa yhtenäisesti. Ohjelmoinnissa vaadittaisiin myös monien eri teknologioiden osaamista. On toisaalta olemassa työkaluja, joilla voidaan kattaa monia alustoja samanaikaisesti. Esimerkiksi Unity-pelimoottoria käyttäessä sama C#-koodi voidaan kääntää tunnetuimmille työpöytä- ja mobiilikäyttöjärjestelmille. Unityssä on myös jatkuvasti kehittyvä tuki kääntää koodi WebGL-pohjaiseksi selaimilla toimivaksi JavaScript koodiksi [61]. Myös Xamarin- ja Qt-sovelluskehykset mahdollistavat saman koodin käyttämisen useammalla alustalla samanaikaisesti [62; 63].

7. YHTEENVETO

Sovelluksen toteuttamiseen valitun Meteor-sovelluskehyksen käyttämä kommunikaatio ei ole paras mahdollinen tilanteissa, joissa paljon viestejä on saatava siirrettyä reaaliaikaisesti. Tilannetta saatiin parannettua hyödyntämällä pelkkää WebSocket-kommunikaatiota käyttävää Streamy-kirjastoa paikkaviestien välityksessä, jolloin saatiin ohitettua tietokantaan tallentaminen ja sen muutosten kuuntelu. Tällöin päästiin hallitsemaan paremmin viestien välitystä, jolloin palvelimelta lähetettäviä viestejä oli mahdollista vähentää yhdistämällä useita viestejä yhdeksi koosteviestiksi. Koosteviestin kokoa voitiin myös olennaisesti pienentää, jättämällä JSON-objektien nimi-arvo-parien nimet ja objektien tunnisteet pois. Meteorin DDP-pohjaista kommunikointia voi kuitenkin käyttää rinnalla vähemmän aikakriittisiin operaatioihin.

Kommunikaation lisäksi ratkaiseva tekijä suorituskyvyssä oli grafiikan piirtäminen. Heikkotehoisemmilla mobiililaitteilla sovelluksessa käytetyn SVG:n piirtäminen on vielä melko raskasta, mutta laitteiden tehot kasvavat jatkuvasti ja uudemmilla laitteilla nähtiinkin nopeampaa piirtoa. Tehdyissä piirtotekniikoiden vertailuissa ainoastaan WebGL-pohjainen renderöinti tarjosi olennaisesti paremman suorituskyvyn. D3.js:n käyttämä transition-ominaisuus auttoi poistamaan hitailla viestienpäivitystaajuuksilla tapahtuvaa nykimistä aloittamalla tietyn kestoisen animoinnin nykytilanteesta päivitettävään tilanteeseen kuvataajuudesta riippumatta hyödyntämällä *requestAnimationFrame*-metodia. Samanaikaisesti liikkuvien objektien määrän kasvaessa canvas ja WebGL-pohjaiset ratkaisut ovat nopeimpia.

Websovellusten käyttämät tekniikat eivät suoraan sovellu pelimäisten sovellusten tekoon, joilla on paljon suuremmat vasteaikavaatimukset kuin perinteisillä websovelluksilla. Esimerkiksi TCP/IP-pohjainen WebSocket ei sovellu aikakriittisiin sovelluksiin huonossa verkossa. Tulevaisuudessa UDP-pohjainen WebRTC-tekniikka voi ratkaista ongelman. Pelejä tai pelimäisiä sovelluksia on syytä toteuttaa käyttäen varta vasten pelejä varten suunniteltuja algoritmeja ja kirjastoja. On myös syytä miettiä, voiko natiivisovelluksella saavuttaa paremman suorituskyvyn, jolloin kuitenkin alustariippumattomuudesta joudutaan tinkimään.

LÄHTEET

[1] A. Juntunen, E. Jalonen, S. Luukkainen, HTML 5 in Mobile Devices -- Drivers and Restraints, System Sciences (HICSS), 2013 46th Hawaii International Conference on System Sciences, pp. 1053-1062.

[2] The World Wide Web Consortium (W3C)., How does the Internet work, verkkosivu. Saatavissa (viitattu 11.3.2016): https://www.w3.org/wiki/How_does_the_Internet_work.

[3] T. Berners-Lee, Information Managent: A Proposal, verkkosivu. Saatavissa (viitattu 11.03.2016): http://www.w3.org/History/1989/proposal.html.

[4] The World Wide Web Consortium (W3C), W3C Technical Report Development, verkkosivu. Saatavissa (viitattu 3.4.2016): https://www.w3.org/2005/10/Process-20051014/tr.html#rec-advance.

[5] T. Olsson, P. O'Brien, CSS Version, verkkosivu. Saatavissa (viitattu 18.9.2016): http://reference.sitepoint.com/css/csshistory.

[6] N. Zakas, Web definitions: DOM, Ajax and more, verkkosivu. Saatavissa (viitattu 22.8.2016): https://www.nczonline.net/blog/2009/09/29/web-definitions-dom-ajax-and-more/.

[7] S. Sonnes, What Does it Mean To "Render" a Webpage? verkkosivu. Saatavissa (viitattu 28.8.2016): http://www.pathinteractive.com/blog/design-development/rendering-a-webpage-with-google-webmaster-tools/.

[8] S. Chapman, The Three Parts of JavaScript, verkkosivu. Saatavissa (viitattu 22.8.2016): http://javascript.about.com/od/reference/a/component.htm.

[9] M. Forsström, JavaScriptin lyhyt historia, verkkosivu. Saatavissa (viitattu 29.7.2016): https://fraktio.fi/blogi/javascriptin-lyhyt-historia/.

[10] J. Garrett, Ajax: A New Approach to Web Applications, verkkosivu. Saatavissa (viitattu 28.8.2016): http://adaptivepath.org/ideas/ajax-new-approach-web-applications/.

[11] Draft: JSX Specification, Facebook, verkkosivu. Saatavissa (viitattu 30.10.2016): https://facebook.github.io/jsx/.

[12] C. Borodescu, Web Sites vs. Web Apps: What the experts think, verkkosivu. Saatavissa (viitattu 29.7.2016): http://www.visionmobile.com/blog/2013/07/web-sites-vs-web-apps-what-the-experts-think/.

[13] A. Deveria, Can I use... verkkosivu. Saatavissa (viitattu 17.4.2016): http://caniuse.com/.

[14] Modernizr Team, Modernizr: the feature detection library for HTML5/CSS3, verkkosivu. Saatavissa (viitattu 17.4.2016): http://modernizr.com/.

[15] A. Barbier, Polyfills and Transpilers : one code for every browser, verkkosivu. Saatavissa (viitattu 22.8.2016): https://medium.com/@alexbrbr/polyfills-and-transpilers-one-code-for-every-browser-abaa85145c9c#.x2mjyo9oj.

[16] C. Shoemaker, HTML5 History: Clean URLs for Deep-linking Ajax Applications, verkkosivu. Saatavissa (viitattu 24.4.2014): http://www.codemag.com/Article/1301091.

[17] C. Peterson, Learning Responsive Web Design, 1st ed. O'Reilly Media, Inc., Sebastopol, California, USA, 2014, 412 p.

[18] S. Gamon, CSS pixel ratios (or, why why all mobile sites are 320 pixels), verkkosivu. Saatavissa (viitattu 5.11.2016): http://gamon.org/blog/2013/04/26/css-pixel-ratios-or-why-all-mobile-sites-are-320-pixels/.

[19] B. Erb, Concurrent Programming for Scalable Web Architectures, verkkosivu. Saatavissa (viitattu 1.10.2016): http://berb.github.io/diploma-thesis/index.html.

[20] D. Kegel, The C10k problem, verkkosivu. Saatavissa (viitattu 2.10.2016): http://www.kegel.com/c10k.html.

[21] Pingdom, A history of the dynamic web, verkkosivu. Saatavissa (viitattu 5.7.2016): http://royal.pingdom.com/2007/12/07/a-history-of-the-dynamic-web/.

[22] P. Garaizar, Benefits and Pitfalls of Using HTML5 APIs for Online Experiments and Simulations, 2012, 2012, .

[23] P. Lubbers, F. Greco, HTML5 Web Sockets: A Quantum Leap in Scalability for the Web, verkkosivu. Saatavissa (viitattu 17.4.2016): http://www.websocket.org/quantum.html.

[24] P. Banks, The State of Real-Time Web in 2016, verkkosivu. Saatavissa (viitattu 30.9.2016): https://banksco.de/p/state-of-realtime-web-2016.html.

[25] E. Li, Optimizing WebSockets Bandwidth, verkkosivu. Saatavissa (viitattu 1.10.2016): http://buildnewgames.com/optimizing-websockets-bandwidth/.

[26] WebRTC - Protocols, verkkosivu. Saatavissa (viitattu 28.8.2016): http://www.tutorialspoint.com/webrtc/webrtc_protocols.htm.

[27] S. Gunneweg, How I mastered WebRTC, verkkosivu. Saatavissa (viitattu 28.8.2016): http://engineering.spilgames.com/mastering-webrtc/.

[28] A. Deveria, Can I use... WebRTC Peer-to-peer connections, verkkosivu. Saatavissa (viitattu 28.8.2016): http://caniuse.com/#feat=rtcpeerconnection.

[29] border-radius - CSS, Mozilla Developer Network, verkkosivu. Saatavissa (viitattu 1.11.2016): https://developer.mozilla.org/en-US/docs/Web/CSS/border-radius.

[30] The World Wide Web Consortium (W3C), SVG Document Object Model (DOM), verkkosivu. Saatavissa (viitattu 22.8.2016): https://www.w3.org/TR/SVG/svgdom.html.

[31] J. Doyle, Transforms: A Complicated Love Story, verkkosivu. Saatavissa (viitattu 27.8.2016): https://css-tricks.com/svg-animation-on-css-transforms/.

[32] D. Rousset, The Complete Guide to Building HTML5 Games with Canvas & SVG, verkkosivu. Saatavissa (viitattu 17.4.2016): http://www.htmlgoodies.com/html5/client/the-complete-guide-to-building-html5-games-with-canvas-svg.html.

[33] T. Mikkonen, A. Taivalsaari, Reports of the Web's Death Are Greatly Exaggerated, Computer, Vol. 44, No. 5, 2011, pp. 30-36.

[34] A. Hallock, HTML5 Game Engines, verkkosivu. Saatavissa (viitattu 29.9.2016): http://html5gameengine.com/.

[35] Web Workers (Windows), Microsoft, verkkosivu. Saatavissa (viitattu 22.8.2016): https://msdn.microsoft.com/en-us/library/hh673568(v=vs.85).aspx.

[36] E. Swenson-Healey, Composing Synchronous and Asynchronous Functions in JavaScript, verkkosivu. Saatavissa (viitattu 1.11.2016): http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/.

[37] The World Wide Web Consortium (W3C), Web Workers, verkkosivu. Saatavissa (viitattu 17.4.2016): http://www.w3.org/TR/workers/.

[38] T. Garsiel, P. Irish, How Browsers Work: Behind the scenes of modern web browsers, verkkosivu. Saatavissa (viitattu 25.9.2016): http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/.

[39] D. Rowinski, What Gives Microsoft Edge In The Browser Wars, verkkosivu. Saatavissa (viitattu 25.9.2016): https://arc.applause.com/2015/01/28/gives-microsofts-new-project-spartan-edge-browser-wars/.

[40] P. Irish, What forces layout / reflow, verkkosivu. Saatavissa (viitattu 25.9.2016): https://gist.github.com/paulirish/5d52fb081b3570c81e3a.

[41] K. Norton, On Layout & Web Performance, verkkosivu. Saatavissa (viitattu 25.9.2016): http://kellegous.com/j/2013/01/26/layout-performance/.

[42] W. Page, Preventing 'layoyt trashing', verkkosivu. Saatavissa (viitattu 25.9.2016): http://wilsonpage.co.uk/preventing-layout-thrashing/.

[43] P. Lewis, Web Fundamentals - Rendering performance, verkkosivu. Saatavissa (viitattu 28.3.2016): https://developers.google.com/web/fundamentals/performance/rendering/.

[44] K. Ghazi, The Gamer's Graphics & Display Settings Guide, verkkosivu. Saatavissa (viitattu 25.9.2016): http://www.tweakguides.com/Graphics_9.html.

[45] J. Edwards, Creating Accurate Timers In JavaScript, verkkosivu. Saatavissa (viitattu 25.9.2016): https://www.sitepoint.com/creating-accurate-timers-in-javascript/#comment-1146899014.

[46] B. Dawson, Windows Timer Resolution: Megawatts Wasted, verkkosivu. Saatavissa (viitattu 25.9.2016): https://randomascii.wordpress.com/2013/07/08/windows-timer-resolution-megawatts-wasted/.

[47] T. Wiltzius, Jank Busting for Better Rendering Performance, verkkosivu. Saatavissa (viitattu 27.7.2016): http://www.html5rocks.com/en/tutorials/speed/rendering/.

[48] What is Meteor ? Meteor Development Group, verkkosivu. Saatavissa (viitattu 5.11.2016): http://docs.meteor.com/#what-is-meteor.

[49] G. Dajaeger, Comparing Angular, Aurelia and React: Is there a next-gen JS framework that rules them all? verkkosivu. Saatavissa (viitattu 27.8.2016): http://www.ae.be/blog-en/comparing-angular-aurelia-react-js-framework/.

[50] S. Stubailo, Optimistic UI with Meteor, verkkosivu. Saatavissa (viitattu 27.8.2016): http://info.meteor.com/blog/optimistic-ui-with-meteor-latency-compensation.

[51] M. Bostock, Working with transitions, verkkosivu. Saatavissa (viitattu 27.7.2016): https://bost.ocks.org/mike/transition/.

[52] M. Bostock, Transition timing could be more predictable and deterministic. verkkosivu. Saatavissa (viitattu 7.11.2016): https://github.com/d3/d3/issues/1336#issuecomment-19858802.

[53] J. Homan, Relational vs. non-relational databses: Which one is right for you ? verkkosivu. Saatavissa (viitattu 30.9.2016): https://www.pluralsight.com/blog/software-development/relational-non-relational-databases.

[54] J. Nielsen, Response Times: The 3 Important Limits, Nielsen Norman Group, verkkosivu. Saatavissa (viitattu 30.7.2016): https://www.nngroup.com/articles/response-times-3-important-limits/.

[55] G. Gambetta, Fast-Paced Multiplayer, verkkosivu. Saatavissa (viitattu 22.8.2016): http://www.gabrielgambetta.com/fast_paced_multiplayer.html.

[56] J. Leicher, Use meteor underlying sockets for realtime communications, verkkosivu. Saatavissa (viitattu 18.9.2016): https://github.com/YuukanOO/streamy.

[57] Canvas, Microsoft, verkkosivu. Saatavissa (viitattu 22.8.2016): https://msdn.microsoft.com/en-us/library/bg124101(v=vs.85).aspx.

[58] MongoDB, Our Biggest Release ever, verkkosivu. Saatavissa (viitattu 22.8.2016): https://www.mongodb.com/mongodb-3.0.

[59] Migrating to Meteor 1.3, verkkosivu. Saatavissa (viitattu 22.8.2016): https://guide.meteor.com/1.3-migration.html.

[60] Migrating to Meteor 1.4, verkkosivu. Saatavissa (viitattu 22.8.2016): https://guide.meteor.com/1.4-migration.html.

[61] Unity - Multiplatfrom - Publish your game to over 10 platforms, Unity Technologies, verkkosivu. Saatavissa (viitattu 1.11.2016): https://unity3d.com/unity/multiplatform/.

[62] Introduction to Mobile Development - Xamarin, Microsoft, verkkosivu. Saatavissa (viitattu 1.11.2016): https://developer.xamarin.com/guides/cross-platform/getting_started/introduction_to_mobile_development/#Introduction_to_Xamarin.

[63] Qt for Application Development, Qt Company, verkkosivu. Saatavissa (viitattu 1.11.2016): https://www.qt.io/qt-for-application-development/.